アスキー・ラーニングシステム　①入門コース

入門 C言語

三田典玄　著

アスキー出版局

アスキー・ラーニングシステム　①入門コース

入門 C言語

三田典玄　著

アスキー出版局

■本文イラスト　久保田 彩子

はじめに

C言語は，ここ数年たいへんな勢いでパーソナル・コンピュータで使われるようになったコンパイラ型の言語です．現在，ワードプロセッサをはじめ数多くのアプリケーションがC言語で実際に書かれており，また多くのユーザーが使っています．

C言語自身は，営利を目的とした企業によって作られたものではなく，研究所レベルで自由に作られ育てられてきた言語の1つです．そのため柔軟な構造を持っており，大型コンピュータからパソコンまで，およそコンピュータと名前が付いている機械の上ならばどこでも使うことができます．このことは，C言語の普及に役立ちましたが，また一方では混乱も起こしました．しかしこの混乱もC言語の誕生から10年を経た今では，そのメリットの大きさ(どんなコンピュータ上でも同じように使える)のほうがデメリットを越えるまでになったといってよいでしょう．ただし，〝メリット〟の実際の効用は，どんなコンピュータでも〝同じかたちでプログラミングが可能〟ということであって，〝まったく同じプログラムが動く〟ということではありません．

プログラミング言語の歴史のなかでC言語が果たした役割は，かなり大きなものです．またC言語を語る上で欠かせない存在となったUNIXは，今では16ビットのパソコンでも動くようになりました．しかし，そのアプリケーションは，ハードウェアの低価格化や機能の進歩に追いついていないのが現状です．この現状を踏まえれば，C言語でアプリケーションを書くということがソフトウェアの生産性を高めるためにいかに重要なことなのかがおわかりいただけると思います．C言語はあらゆるパソコンでかなり高い互換性を保って実際に動いている言語なのです．

本書を書く上でとくに重点を置いたのは以下の点です．

1．覚えることがらを少なくし，できるだけはやくプログラムが書けるように配慮した
2．基本的な概念が理解できるような構成を心がけた
3．C言語をより深く理解するために，コンピュータの基本的な構造についての解説を行った

すでにC言語をある程度修められた方々にとっては，本書がC言語のより深い理解の助けとなれば幸いです．そして何よりも初心者の方には，本書がC言語学習のきっかけとなれば筆者にとってこれに勝る喜びはありません．

1986年2月23日　自宅マシン・ルームにて　　　三田　典玄　(asc11604/dwq@uupun. juice)

C言語ラーニングシステム　全3巻の構成

C言語のアスキー・ラーニングシステムは，入門C言語，実習C言語，応用C言語の全3巻で構成されます．

従来のC言語の書籍は，主にコンピュータをよく知っている人たちを対象に書かれており，数多くのことがらが1冊の本につまっていました．このため基本的なコンピュータの知識を持っていない初心者が学習するにはかなり無理があったように思います．また逆に，コンピュータを使いこなしている上級者にとっては，細かい説明やくわしい内部構造など，ほんとうに知りたいことがいまひとつ載っていないことへの不満が少なからずあったことでしょう．

今回のシリーズでは，初心者から無理なく学習でき，また上級者にも満足できるようにとの主旨から全体を3巻にわけて構成しました．以下に各巻の概要を紹介しておきます．

入門C言語：本巻では，C言語で簡単なプログラムが書けるようになることをテーマとしています．また，C言語を学ぶ上で不可欠なコンピュータの基本構造についても解説し，この先より大きなプログラムが書けるようになるための基礎的な力をつけることを主眼に置きました．

実習C言語：入門編で取り上げなかった「構造体」，「共用体」などの学習と，C言語についてのすべての項目がわかるように体系的な整理を行います．また，各項目ごとに数多くの例題を挙げ，各自でプログラムを組んでいく際にマシンのそばに置いていつでも参照できるように構成します．

応用C言語：C言語で実践的なプログラムを記述するために，さらに高度なプログラミングと開発環境について学習します．とくにC言語が使われることの多いMS-DOSやUNIX上でのプログラミングや，従来アセンブラで書かれていたハードウェア割り込みのC言語による記述，オリジナルライブラリの作成，日本語処理のノウハウなどを紹介します．

本シリーズを読むに当たっては，実際にコンピュータを目の前に置いて動かすことを前提としていますが，書籍を読むだけでも体験的な学習ができるように配慮しています．

目　次

第1章 コンパイラとしてのC言語

C言語というプログラミング言語は，ごく最近になって話題にのぼり広く使われるようになった言語です．コンピュータといえばパソコンを思い浮かべることの多くなった今日では，知名度ナンバーワンの言語といえば，やはりBASICということになるでしょう．BASICは，〝ちょっとコンピュータを使ってみる〟という目的にはなかなか便利な言語です．最近は，このBASICもかなり高い機能を持つようになり使い勝手も向上して，できあがったプログラムの実行速度も一昔前のオフコンを凌ぐことさえあります．

しかし，BASICが広まれば広まるほど，その欠点も多くの人たちに知れ渡ることになります．つまりBASICの普及は，そのまま「BASICの次の言語」の模索へとつながっていったわけです．

BASICの次の言語の模索には，やはりBASICの欠点の認識が不可欠です．以下にそれを列挙してみたいと思います．

1．実行速度が遅い
2．構造化されていないので，大きなプログラムが組みにくい
3．プログラムが目に見えるリストの形で動いているので容易に解読されてしまい，機密保持が困難

これらの欠点のほとんどは，BASIC自身の問題というよりも，BASICという言語がインタープリタであることによるものばかりです．最近はBASICコンパイラというものもありますので，これらの欠点はだいぶ緩和されてきました．しかし，BASIC自身がコンパイラに適した言語ではないため，実際に実行されるプログラムのサイズがほかのコンパイラとは比べものにならないくらい大きなものとなってしまい，実用にはいま一歩といった感じは拭えません．

ここでは，この〝コンパイラとはいったいどんなものなのか〟ということから説明することにしましょう．

1.1 インタープリタとコンパイラ

「インタープリタ」や「コンパイラ」といった名称は，たとえば「BASIC」や「FORTRAN」のように言語の書き方(書式)に付けられた名前ではなく，プログラムの実行に至るまでの過程(実行形式)に付けられた名称です．プログラミング言語には数多くの種類がありますが，これらは2つの実行形式のどちらかに分類することができます．

以下にこの2つの過程の違いをくわしく説明しましょう．

■インタープリタとは

まず，インタープリタの代表として，やはりBASICを考えてみましょう．BASICでプログラムを書く場合には，その機能として備わっているエディタを使っています．こんな話をすると，ちょっと疑問に思ってしまう人がいるでしょう．たしかにそれは当り前です．BASICを起動すると，エディタという1つの環境のなかでプログラムを組んだりコマンドを実行したりできるので，その存在をあまり意識しないのが普通です．

そこで，BASICの中身をもっとくわしく見てみましょう(**図1-1**)．

この図のようにBASICと呼んでいるものは，実はいくつかのプログラムの集まりなのです．それぞれのプログラムについて，簡単に説明しておきます．

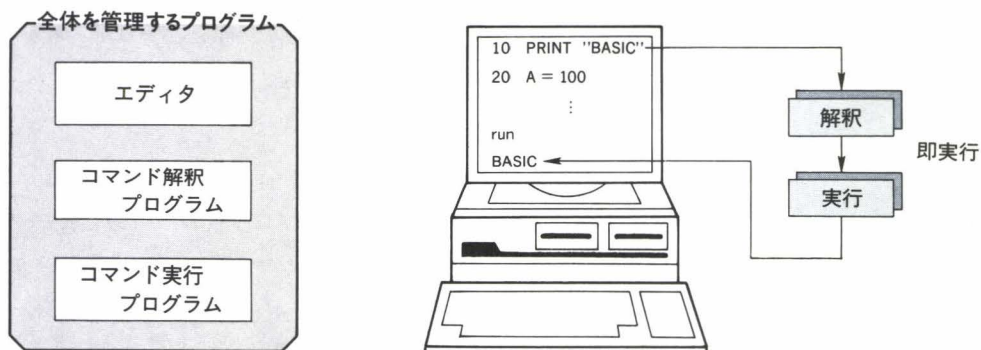



図 1-1 BASICの構成

1．エディタ

行番号を付けたBASICのプログラムを編集し，記憶しておく

2．コマンド解釈プログラム

キーボードから入れた「LIST」や「RUN」などのコマンドやプログラムを解釈する．解釈された命令は3に引き渡され実行される

3．コマンド実行プログラム

2で解釈されたプログラムを実行する．たとえばPRINT文ならば指示された文字や数字を画面に表示する

4．管理プログラム

1～3のプログラムの動きを制御する

このなかの2と3が「インタープリタ」と呼ばれる言語の中心です．つまり，インタープリタは，1つの命令を解釈しては実行するという作業を繰り返しているのです．インタープリタの長所は，以下のとおりです．

1．記述したそのままのプログラムが実行されるので，どこでプログラムが止まってもその場でデバッグができる

2．保存しておくプログラムの大きさが小さくて済む

欠点については，この章の前書きでも述べたとおりですが，一言でいえば「遅い」ということです．

■コンパイラとは

コンパイル(Compile)とは本来〝寄せ集める〟という意味ですが，プログラミング言語の世界では〝コンピュータにわかる言葉に翻訳する〟という意味を持っています．そして，コンパイルを行うプログラムを**コンパイラ**と呼びます．

コンパイラは，人間の読めるプログラムを一度に翻訳して，コンピュータが読める(実行できる)プログラムに変換します．ここで，人間が読めるプログラムを**ソースプログラム**，コンピュータが実行できるプログラムを**実行形式のプログラム**といいます．つまり，すべての命令に対して「解釈」を先に行ってしまい，その結果をどこか別の場所にとっておいて，後で一気に「実行」します．これはインタープリタが，1つの命令ごとに解釈と実行を繰り返していたのとは大きな違いです(**図1-2**)．

完成したプログラムを実行する際には，プログラムがそのまま読めるかどうかということは関係ありません．逆にいえば，1つの命令を実行するごとにプログラムを止めるわけにはいかないので，〝デ

バッグが大変〃ということにもなります．しかし，実行時には，解釈の時間を必要としないので非常に速い速度で実行が行われます．また，実行形式のプログラムだけにしておけば，なかを読むことはむずかしいので秘密の保持にも役立ちます．

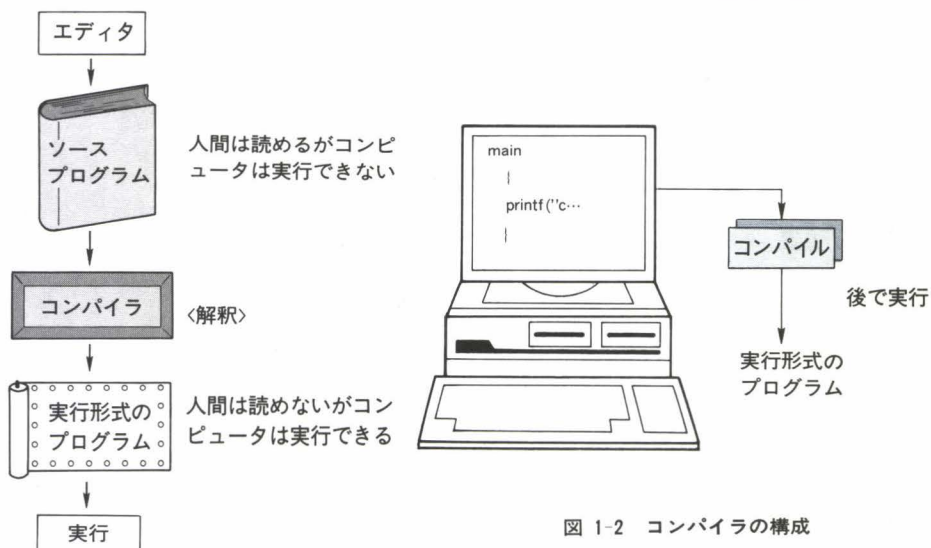



図 1-2 コンパイラの構成

■インタープリタとコンパイラのまとめ

こうして見ると，コンピュータを動かす方法の根本的な考え方の違いが，「インタープリタ」と「コンパイラ」の違いということになります．

歴史的には，まずコンパイラが作られその後にインタープリタが出てきました．どちらも一長一短があり，それぞれ特色を持っています．この本で扱うC言語は，コンパイラとして作られましたが，最近ではインタープリタもあります．また逆に，BASICはインタープリタとして作られたにもかかわらずBASICコンパイラも存在します．どちらもコンパイラ(インタープリタ)ではあまりよくなかった部分を，インタープリタ(コンパイラ)で補おうとして作られたものです．

よく，〃インタープリタは亀，コンパイラは兎〃などという一方的なたとえ話を聞きますが，本来は適材適所で使われるべきものです．インタープリタ言語の〃手軽さ〃にはコンパイラ言語はかないませんし，コンパイラ言語の〃速度〃にはインタープリタはかないません．

1.2　C言語の誕生

本書で扱うC言語は，コンパイラ型の言語です．この〝C〟という奇妙な名前はどこからきたのでしょうか．実は，〝B〟という言語の次にできた言語だからというのがその理由です．しかし残念ながら，B言語の前はA言語ではありません(ちなみに，A言語というと，最近話題の「Ada（エイダ）」のことを指すことが多いようです)．

■C言語の歴史

C言語のルーツは，Martin Richards（マーチン リチャード）が開発した「BCPL言語」にさかのぼります．この言語は，

1．構造化言語と呼ばれる制御構造と制御文を持っている
2．計算機自身の持つデータ型に近いデータ型を持っている
3．ポインタが扱える

などのC言語の持つ特徴のほとんどを備えていました．このBCPLをもとにして作られた言語が「B言語」と呼ばれる言語で，Ken Thompson（ケン トムプソン）によって1970年に書かれました．

B言語は，DEC社のPDP-7というミニコン用のオペレーティング・システムの記述のために考案された言語でした．このPDP-7は，現在のパソコンと比べるとかなりパフォーマンスが落ちますが，それでも当時としては最新鋭のミニコンでした．このB言語で記述されたオペレーティング・システムで最も有名なのは，現在C言語と切っても切り離せない関係にある「UNIX（ユニックス）」です．

C言語はこのB言語の後を受けて，1972年にやはりDEC社のPDP-11というミニコン上のUNIXシステムを記述するために書かれたのがその歴史の始まりです．制作者はDennis（デニス）．M．Ritchie（リッチー）とBrian（ブレイン）．W．Kernighan（カーニハン）です．

図1-3に，C言語を含めたプログラミング言語の変遷をまとめておきます．

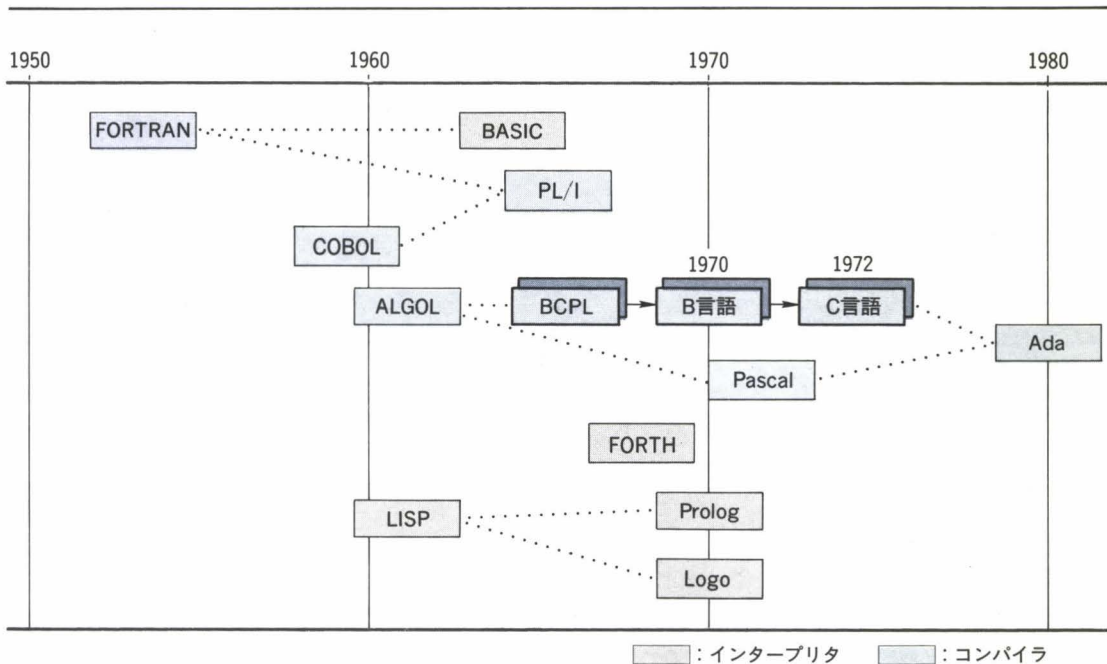



図 1-3 プログラミング言語の変遷

■ C言語の特徴

このような歴史を持ったC言語は，当然のことながらDEC社のミニコンのアーキテクチャ(作り方)や命令セット(機械語)の影響をかなり強く受けています．そして，現在のパソコン(とくに16ビットマシン)は，このミニコンの構造とかなりよく似ています．というより，現在のマイクロプロセッサの構造が，基本的にミニコンをお手本にしているのです．これが，最近のパソコンでC言語が使われるようになってきた理由の1つです．

さて，ミニコンやパソコンなどの比較的小さいコンピュータシステムで動き出したC言語は，IBMなどのメインフレーム上にも乗せられました．どちらかというと保守的なメインフレーマ(†メインフレームを操る人)もC言語の移植性の良さや言語自身のシンプルな構造が，現時点ではともかく将来はメインフレームの分野にもなんらかの影響を与えるに違いないと考えたのだと思います．

† メインフレーム：大型コンピュータのこと

このようにC言語を使うことによって，異なったコンピュータ同士でのプログラムの**互換性の問題**がかなり解決されました．また，機械語で記述され，書いた本人以外はわからないようなプログラムが減りました．とくにオペレーティング・システムの記述などに，C言語は絶大な威力を発揮します．UNIXは，C言語があったために世界中に広まったといっても過言ではありません．

■C言語のマニュアル

最近は少なくなりましたが，C言語のコンパイラを買うとやたらと薄いパッケージが届いて，マニュアルには〝C言語自身のリファレンス・マニュアルとしてB.W.Kernighan/D.M.RitchieのC言語の本を読んでくれ〟と書いてあることがよくありました．つまり，C言語を作った人たちの本が，リファレンス・マニュアルになっているのです．C言語は，それだけ統一性，互換性のある言語だといえます．そしてほんとうの「標準」といえるのは，やはりUNIX上のC言語でしょう．

■各社のC言語の違い

最近のパソコンでは，どのオペレーティング・システム上でも，かならずといってよいほどC言語を走らせることができます．8ビット系では「CP/M」や「MSX-DOS」，16ビット系の「MS-DOS」，「CP/M-86」，IBM-PC用の「PC-DOS」などのOS上に，それぞれ何種類ものC言語が存在します．とくに最近のマイコン機器開発用のシステム上では，かならずC言語が動いています．

そして，それらはかなりの互換性を持っています．つまり，どのオペレーティング・システム上のものでも，使う立場からすれば〝ほとんど同じもの〟として扱えます．もちろん，C言語を使い込むことによって，それぞれの小さな違いを見い出すことはできます．

もしあなたがC言語コンパイラの購入を考えているのならば，複数の種類のC言語を使ったことがある人にアドバイスを受けることをおすすめします．なぜかというと，各社のC言語の違いというのは，普通の基準からすると細かい部分なので，かなり使い込んでからでないと判断がむずかしいからです．ましてや仕事でC言語を使う場合は，その小さな違いが仕事の可否や開発効率にかかわってくるかもしれません．

1.3　C言語の長所と短所

プログラムの開発にあたって，開発者がしなければならない作業の1つは，開発を行うプログラミング言語の選択です．

現在あるプログラミング言語は，それぞれが特徴を持ち得意な分野を持っています．C言語にも，やはり「得意，不得意」があり，それがそのままこの言語の長所と短所になっています．

以下にその長所と短所を考えてみます．

■C言語の長所

C言語は，次のような長所を持っています．

1．プログラムにあたって，覚えることが非常に少ない．また，例外もあまりない
2．構造化された制御構造を持っているので，作業の手順化，明確化が容易に可能
3．言語構造自体がシンプルなので，比較的小さいメモリや外部記憶装置しかないハードウェアでも開発が効率よく行える
4．ビットの操作など，きめの細かい処理が可能
5．構造体や共用体などの複雑なデータ型を扱えるので，プログラム自身を簡潔に記述できる

■C言語の短所

反対に短所としては次のような点が挙げられます．

1．行を意識しない記述が可能なため，プログラマ自身が気をつけておかないと読みにくいプログラムができてしまう
2．プログラムのモジュール化を心がけていないと，わかりにくいプログラムになることが多い
3．下手なプログラムを書くと，プログラム・エラーがなくても暴走することがある

長所と短所というのは，同じ本質を違う方向から見ているために起こる表現の違いと考えられます．じっくりとこれらの長所，短所を眺めると，C言語の本質が見えてきます．そして，〝C言語は，ほんとうにこのプログラムを書くのに向いているのだろうか〟と，まず考えて欲しいものです．

1.4　C言語の使用環境を整える

使用環境と一口にいっても，最終的にできるものの形によってこれらの環境はかなり違います．たとえば「オペレーティング・システム上で動くプログラム」が欲しいのか，「制御機器のROM化されたプログラム」が欲しいのかによってその開発環境は変わってきます．ここでは，一般的なオペレーティング・システム上で動くプログラムを開発することを目的に話を進めていきます．

■オペレーティング・システムの意義

これまで何度も「オペレーティング・システム」という言葉が出てきました．〝その名前はよく耳にするが，実際なんのためにあるのかよくわからない〟という人もいるでしょう．ここでは，具体的な使用環境の説明にはいる前に，オペレーティング・システムの意義を解説しておきます．

オペレーティング・システム(OS)とは，人間にとってわかりにくいコンピュータの操作をより使いやすくするために生まれたプログラムのことです．たとえば，〝キーボードから文字を1つ読んでくる〟という場合を考えてみましょう．もし，使いたいマシンがまったくの新製品でBASICも何もなければ次のようなプログラムを機械語で書いてやらなければなりません．

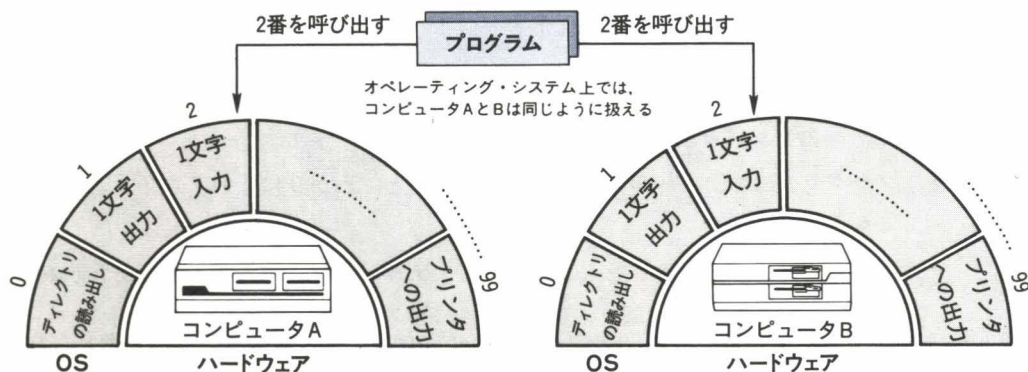



図 1-4　オペレーティング・システムの環境

1．キーボードを見にいく
2．何も入力されていなければ，決められたメモリに"データがない"という印を付ける
3．入力されていたら，決められたメモリに"データあり"の印を付ける．そして，入力されたデータを別の決められたメモリに格納する
4．このプログラムを呼び出したプログラムに戻る

このようなプログラムは，ワードプロセッサを書く場合にはそれこそ何万回と呼び出されるものです．そしてこういった基礎的なプログラムは，どんなプログラムを記述する際にも必要ですから，コンピュータが違っても同じように利用できると便利です．

これらの要求に答えて作られたのがオペレーティング・システムという"一般によく使われるプログラムの集まり"なのです．そして同じオペレーティング・システム上では，どのコンピュータでも"こうすればキーボードから1文字入力できる"という規格が決まっています(**図 1-4**)．

つまりプログラマは，このような基礎的なプログラムを書く必要はなくどのマシンでも同じ扱いでプログラムを書けばよいことになります．そして作成したアプリケーション・プログラムは，OS という「共通規格」があることで，より多くのコンピュータで動かすことができます．

C言語も基本的にオペレーティング・システム上で動くアプリケーション・プログラムの1つです．これに対して BASIC は，どうでしょうか．BASIC は通常 OS 上に載っているわけではありません．

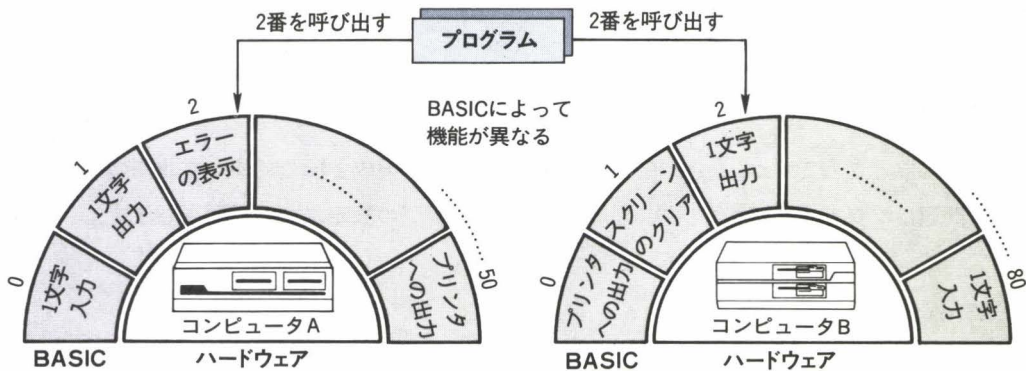



図 1-5　BASICの環境

実は，BASICはそれ自身にオペレーティング・システムと同じような環境を取り込んでしまっているのです．しかし，それは機種によって異なっているため同じプログラムでもマシンが違うと動きません(図1-5)．

オペレーティング・システムという考え方は，大型コンピュータからきたものです．現在私たちが使っている「MS-DOS」とか「CP/M」は，みな大型のそれを手本として作られています．

また，オペレーティング・システムがもたらすものとして忘れてはならないことに，作業環境(ユーザー・インターフェース)の向上があります．私たちは，「DIR」とか「COPY」などのコマンドを用いて仕事を進めますが，これはOSがディスクとのやりとりやファイルの扱いなどのめんどうな作業を管理してくれるために，利用するユーザーは内部で何が行われているのかをいっさい気にせずに済むのです．

■必要なハードウェアとソフトウェア

さて，C言語を使うにあたって，必要なハードウェアとソフトウェアを以下に示します．

<ハードウェア>

・コンピュータ本体：オペレーティング・システムの動くコンピューター式
・外部記憶装置　　：少なくともフロッピーディスク2台
・プリンタ装置　　：なくてもよいが，あると便利

<ソフトウェア>

・オペレーティング・システム
・C言語コンパイラ一式
・エディタ
・リンカ(たいていはオペレーティング・システムやコンパイラに付属している)

ざっとこんなところですが，ハードウェアは通常の必要な構成と同じですから説明の必要はないでしょう．問題はソフトウェアです．次に，このソフトウェアの設定について解説します．

■ソフトウェアの準備

ここでは，ソフトウェアをとりそろえるための一般的な準備について解説していきます．具体的なディスクの準備や環境の設定は，使用するオペレーティング・システムとC言語によって異なりますので，巻末のAPPENDIXを参照して準備を行ってください．

◆ドライブAの内容

ドライブAには，**C言語コンパイラ**，**リンカ**，**エディタ**など必要なファイルをすべて収めます(**図1-6**)．

- C言語コンパイラ
  複数のプログラムから構成されることが多い
- リンカ
  C言語に付属していない場合は，OSのシステム・ディスクからコピーする
- エディタ
  ユーザーが個別に用意
- 「.H」と「.LIB」という拡張子が付くファイル群
- コンパイル用のバッチファイル
  必要な場合作成する

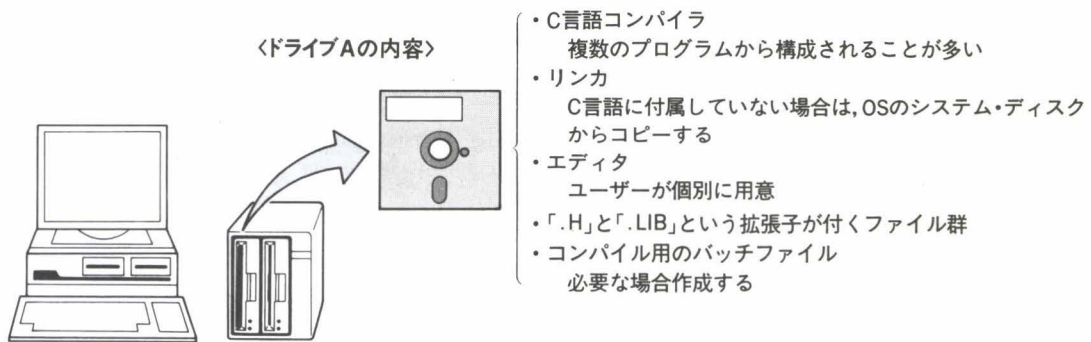



図1-6　ドライブAの内容

C言語コンパイラは，1つのプログラムではなく通常はいくつかのプログラムから構成されます．C言語のシステム・ディスクには，コンパイラ本体のほかに「.H」や「.LIB」という拡張子が付いたファイルが入っていますが，これらもコンパイルを行う際に必要なものです．その他リンカと呼ばれるプログラムが付属していない場合は，オペレーティング・システムのディスクからコピーをしておきます．また，コンパイルためのコマンドをあらかじめ作成しておくと便利なC言語もあります．これについては後でもう一度触れます．

もう1つ用意するものは，エディタと呼ばれるC言語のプログラムを書くための道具です．BASICには言語自体にエディタの機能が含まれていましたが，C言語のプログラムを書く場合には別に用意する必要があります．本書ではエディタの具体的な操作については触れませんが，プログラムを書く際の手足となるものですから，その操作には十分に慣れておいてください．

以上のファイルを入れるとかなりの容量になるので，1MBタイプのフロッピーディスクでなければ入らないことがあります．容量の少ないフロッピーを使っている人は，「.H」や「.LIB」が付くファイルをドライブBに置くなどの対処をします．

UNIXやMS-DOSのような「階層ディレクトリ」を持つOSでは，この機能を利用してファイルの属性ごとに整理しておくと，後で管理がしやすくなります(**図1-7**)．

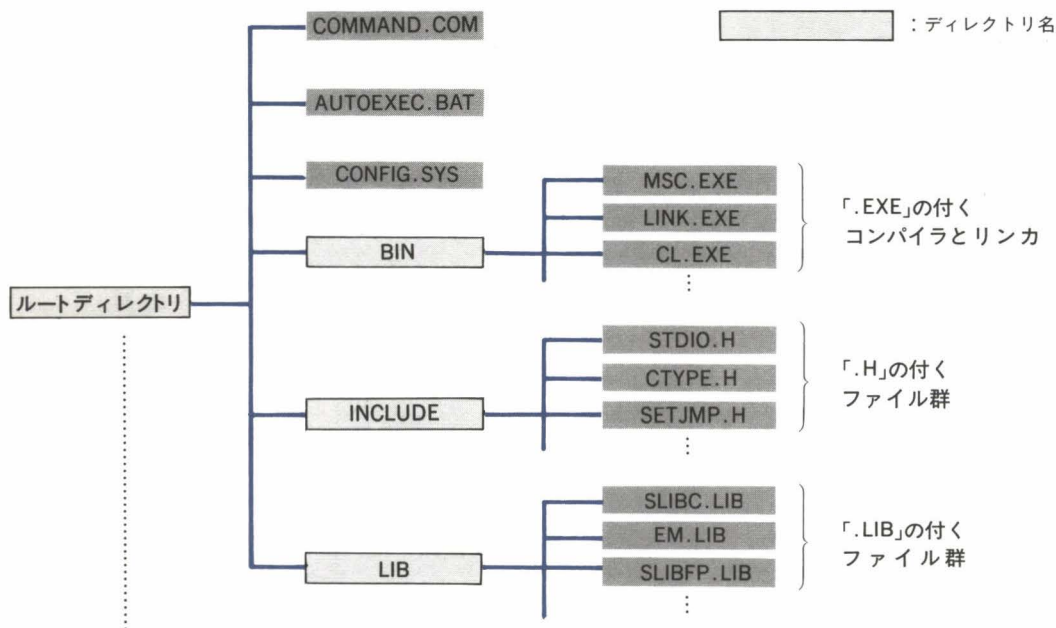



図1-7　階層ディレクトリの概念図

◆ドライブBの内容

ドライブBには，エディタで作成したC言語のソースファイルとそれをコンパイルしてできた結果を置くことにします(**図1-8**)．

〈ドライブBの内容〉

・C言語のソースファイル(プログラム)
・コンパイルの結果できた実行形式のファイル

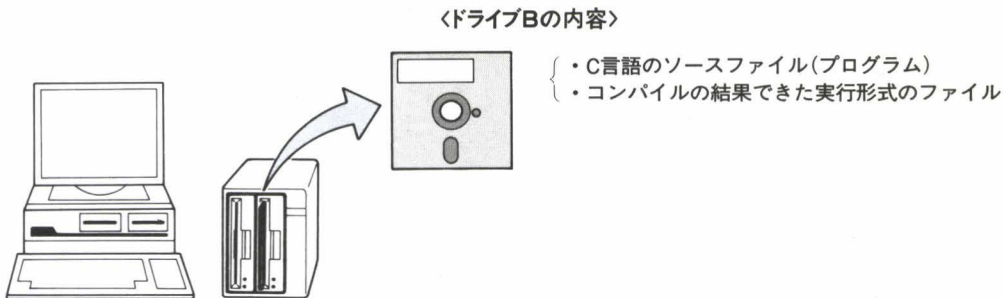



図 1-8 ドライブBの内容

◆オペレーティング・システムの環境の設定

MS-DOSの場合は，システムを起動する際にユーザーが指定した環境にあらかじめ設定を行う機能があり，これを利用すると便利です．これには，オープンするファイルやバッファの数などを指定する「CONFIG.SYS」と入力されたコマンドを探すディレクトリ(PATH(パス)といいます)などを指定する「AUTOEXEC.BAT」の2つのファイルを用います．以下に代表的なファイルの設定例を示します．

〈CONFIG.SYSの内容〉

```
files = 20          ……一度に扱うファイルは20個
buffers = 20        ……ファイルバッファは20個
break = on
shell = a:¥command.com a:¥ /p
```

〈AUTOEXEC.BATの内容〉

```
echo off
path = a:¥;b:¥     ……コマンドをサーチする場所(ここではドライブAとドライブBのルートディレクトリになる)
set INCLUDE=a:¥include  ┐
set LIB=a:¥lib          ├ コンパイラが使用するディレクトリの設定(各コンパイラによって異なる)
set TMP=b:¥             ┘
```

図 1-9 CONFIG.SYSとAUTOEXEC.BATの設定例

◆コンパイル用のコマンド

コンパイルという作業では，通常多くのプログラムを次々に起動したりほかのファイルを読み込んだりということが行われます．そのために，あらかじめその作業手順を記述したバッチファイル(CP/M系のOSではサブミットファイル)を用意しておくと便利です．これは，「CC.BAT」(CP/Mでは「CC.SUB」)というファイル名で作成します．そしてこのファイルは一度作っておくと，後はコマンドと同じように扱うことが可能です．例として，バッチファイルとそれがどのように実行されていくかを示します(**図1-10**)．

```
A>type cc.bat ↵ ………コンパイル用のバッチファイルの内容を表示する(Microsoft C Compilerの場合)
msc %1.c, %1.obj;
link %1.obj, %1.exe, NUL, em.lib, slibfp.lib, slibc.lib }実行形式のファイルを作るまでの手順
              ┌ファイル名
A>cc b:test ↵ ………バッチファイルの実行

A>msc b:test.c, b:test.obj;………………「%1」のところに指定したファイル名が置き換わって自動実行される
Microsoft C Compiler Version 3.00
(C)Copyright Microsoft Corp 1984 1985

A>link b:test.obj, b:test.exe, NUL, em.lib, slibfp.lib, slibc.lib

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

A> ………実行の終了
```

図 1-10　コンパイル用のバッチファイルの実行

なお，C言語によっては，コンパイルのための専用コマンドが付属してくるものもあります．この場合には，ファイル名の指定方法などが異なることもありますので注意してください．

1.5 作ったプログラムをコンパイルして実行させるまで

前節で，C言語のプログラムを書いて実行するための環境はすべて整いました．ここでは，プログラムの作成から実行までの手順を示すことにします．なお，2章以降では，C言語プログラムと実行結果のみを掲載しますので，ここで解説する手順でプログラムの作成と実行を行ってください．

■エディタによるプログラム(ソースファイル)の作成

まず，C言語のプログラムを書かなければなりません．これは，「エディタ」というプログラムを起動して行います．ここでは，MS-DOSのシステム・ディスクに付いてくる「EDLIN」を使ってプログラムを記述する例を紹介します(**図1-11**)．

```
              ┌──── ファイル名にはかならず「.C」を付ける
A>edlin test.c ↵ ………エディタ(EDLIN)の起動
新しいファイルです………エディタからのメッセージ
 *i ↵ ……………………………入力モードに入る
     1: main() ↵
     2:     { ↵                          ライン・エディタなのでカーソルは
     3:     printf("C Language"); ↵      1行の範囲でしか移動しない
     4:     } ↵
     5: ^c ………CTRL + C で入力モードの終了
 *e ↵ ……………………………ファイルをディスクに書き込み終了する

 A>   └── 行番号は自動的に付けられる
```

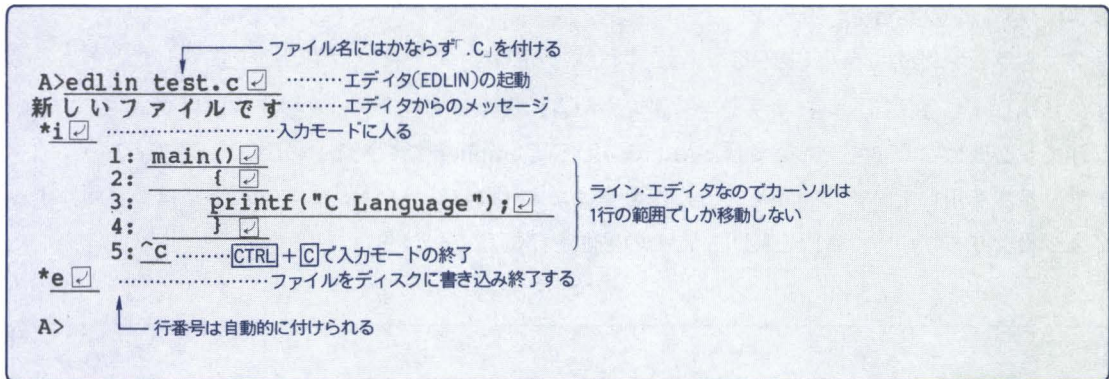



図 1-11 エディタによるプログラムの作成

EDLINは行単位で編集を行うライン・エディタですが，BASICのエディタに慣れている人は，画面単位でカーソルが移動するスクリーン・エディタを用意するとよいでしょう．

なお，作成するプログラムのファイル名にはかならず「.C」という拡張子を付けておきます．これを付けておかないと以後のコンパイルという作業が行えません．

■作成したファイルの確認

エディタで作成したプログラムは，オペレーティング・システムの「TYPE」コマンドで確認をします(**図1-12**).

```
A>type b:test.c ↵ ………エディタで作成したソースファイルの内容を表示する

main()
    {
    printf("C Language");
    }

A>
```

図 1-12　作成したプログラムの確認

■コンパイルの実行

プログラムを作成するとBASICならば，すぐに「RUN」としてプログラムを実行したいところです．しかし，C言語はコンパイラ型の言語ですから，実際にプログラムを動かす前にコンパイルという作業が必要になります．ここでは，Microsoft C Compilerでさきほどの「CC.BAT」というバッチファイルを用いてコンパイルを行った例を挙げます(**図1-13**)．なお，この実行方法はC言語によって多少異なりますので，APPENDIXでその手順を確認してください．

図 1-13　コンパイルの実行

実行が終了して，再びプロンプト(A＞)が現れるまで，しばらく時間がかかります．このプロンプトが表示された後で次の作業に進みます．

■エラーが出た場合の対処

図1-13の例ではコンパイルは成功しましたが,プログラムに誤りがあると画面上に**エラーメッセージ**が現れます．とくに初心者のうちは，エラーが山のように出るとびっくりするものですが，このエラーメッセージには見方があります．具体的な方法は6章で取り上げていますので，エラーで先に進まなくなってしまったらその章を開いて見てください．

この場合には，エディタに戻ってソースファイルの誤りを修正し，もう一度コンパイルという作業を繰り返します．

■プログラムの実行

無事にコンパイルが終了すると，いよいよ実行です．まず，ドライブBに実行形式のファイルができあがっていますので，それを確認してみましょう．

図 1-14　コンパイルによって作成された実行ファイル

図1-14にあるファイルのうち，[†]「TEST.EXE」が実行形式のいわば〝動くプログム〟です．「TEST.OBJ」は，実行ファイルができるまでに作られた中間ファイルです．

さて，やっと動くプログラムが手に入りました．このプログラムは，普段使っているコマンドと同じように実行することができます(**図1-15**)．

---

† CP/M-86では「TEST.CMD」，CP/Mでは「TEST.COM」となります．

```
A>test ↵ ………プログラムの実行
C Language ……実行結果
A>……………………実行の終了
```

図 1-15　プログラムの実行

■ 実行結果の確認

最後にしなければならないのが，実行結果の確認です．ここでは，プログラムが決められたとおりにきちんと動いているかどうかを確かめてください．正しく動作しない場合はエディタに戻ります．また，コンパイルは成功しても実行時にエラーが起き，コンピュータが**暴走**(どのキーを押しても反応しない状態)することがあります．その場合には，リセットを押してマシンを起動しなおさなければなりません．この場合の具体的な対処法も6章に示します．

■ コンパイル手順のまとめ

以上のような手順で，C言語プログラムの作成から実行までを行っていきます．BASICなどのインタープリタに慣れた人は，このコンパイルという作業はめんどうに思うかも知れません．しかし，この作業には大きなメリットがあるのです．コンパイルについてのくわしい解説は6章に譲ることにしましょう．

最後に，プログラムの作成から実行までの手順を**図1-16**に整理しておきます．

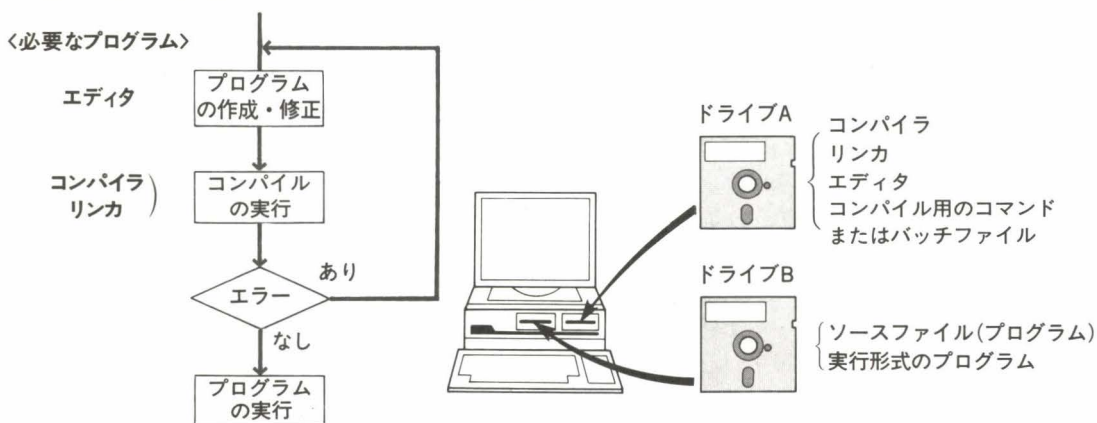



図 1-16　プログラムの作成から実行までの手順

第2章
最も簡単なC言語のプログラミング

どんなプログラミング言語を学習するときも，まず最初に出てくる例題というのは決まっています．それは「ある特定の文字を画面に出力させる」というものです．この本でも，まずはこの例題を取り上げます．

C言語は前章でも説明したとおりコンパイラ型の言語ですから，プログラムを書いてからコンパイルという作業があり，その後に初めて動くプログラムができあがります．この章ではこれらの過程の説明は行いません．また，プログラムを作成し動かす環境としてのハードウェアとソフトウェアは，前章で整っていることとします．

ここではC言語そのものに慣れると同時に，BASICのようなインタープリタ型言語とはまた違ったコンパイラ型言語の〝感じ〟をつかみとっていただけたらと思います．この章でのC言語プログラミングとオペレーションは次章への大事な足掛りとなりますから，しっかり覚えておいてください．

2.1 画面への出力 ―printfの使い方―

画面への文字の出力は, printf という関数によって行われます. 関数については後の章で取り上げますが, ここでは BASIC のコマンドに当たるものと考えてください. この節では, C言語プログラムと BASIC プログラムの対比を行いながら話を進めますので, とくにその書式の違いなどを理解できると思います.

■C言語プログラムのスタイル

C言語のプログラムは, BASIC のプログラムの書式とはかなりスタイルが異なります. まず, 以下の**リスト 2-1** ような2行の文字列を出力するためのプログラムを見てみましょう. これは前章で説明したエディタで書いた「ソースファイル」です.

＊プログラムは小文字で書く

```
1: main()  ………実行単位の名前
2:     {
3:     printf("Line No.1");
4:     printf("Line No.2");
5:     }
```

|→TAB←|

{ }で囲まれた部分がプログラムの本体

セミコロンを忘れないこと

MSX-C, LSI C は, 1行目の前に「#include <stdio.h>」を追加

リスト 2-1 文字列を表示するC言語プログラム

このプログラムは, BASIC で書き直せば**リスト 2-2** のようになります.

```
100 PRINT "Line No.1"
200 PRINT "Line No.2"
300 END
```

リスト 2-2 文字列を表示するBASICプログラム

この2つのプログラムの書式の違いを見てみましょう．

まず気づくことは，C言語のプログラムがすべて小文字で書かれている点です．これはC言語のプログラムを書く上での〝約束事〟の1つです．これまでBASICを使ってきた人のなかには，プログラムを大文字で書く癖のついている人がいるかもしれませんが，C言語では小文字を使うようにしてください．

また，BASICのプログラムにはかならず行番号が必要ですが，C言語では付けません(本書では，リストを見やすくするために後から行番号を付けて掲載しています)．

次にこの2つのプログラムの内容を見比べてみます．ここでBASICのPRINT文に当たるものがC言語では「printf」になっていることがわかるでしょう．それでは，1行目にある「main( )」というのは何でしょうか．これは，このプログラムを実行する単位(**実行単位**)の名前です．そして，その実行単位の範囲を示すために〝{ }〟で囲んで，プログラムの本体がそのなかに書かれています．

BASICではプログラムの実行が基本的に行単位で行われるため，PRINT文だけでこのまま実行させることができます．しかしC言語では，文字を出力するprintf文を実行するために，その器である〝main〟という入れ物をかならず用意する必要があります．これは，C言語がコンパイラであるためにプログラムの最初と終わりを明確にしておかないと，一度にプログラムを変換して実行形式のファイルを作成することができないからです．

また，各行の終わりにある「;」(セミコロン)は，実行文の区切りとして使われています．BASICでは，行か「:」(コロン)で区切られた文ごとに実行の単位が決まりますが，C言語の場合は「;」を使用します．これを忘れると，コンパイラがうまく解釈してくれません．

C言語のプログラムは行という単位に依存しないので，**リスト2-3**のように記述することもできます．

```
1: main(){printf("Line No.1");printf("Line No.2");}
```

1行で書いてしまうこともできる

```
1: main(){
2: printf("Line No.1");
3:
4:
5:                      printf("Line No.2");
```

3:～4: 空行は読みとばしてくれる

5: 空白も自由に取ることができる

リスト 2-3　読みにくいC言語のプログラムの例

このように自由な書式でプログラムを記述することを**フリーフォーマット**といいます．しかし，逆にいえば**リスト 2-3** のような読みにくいプログラムも実際に書けてしまうので注意しなければなりません．本書で掲載するプログラムは，読みやすさを考慮してすべて同じフォーマットで統一してあります．これ以外にもプログラムの書き方はありますが，どの場合でも読みやすいかどうかという点が大切なポイントです．

■改行をするには

さてそれでは，**リスト 2-1** のプログラムをコンパイルして実行させてみましょう．以後，C言語で書かれたソースファイル名は「TEST.C」としておきます．このファイルをコンパイルすると†「TEST.EXE」という実行ファイルができあがります．

実行結果は**図 2-1** のようになります．

```
A>test⏎ ………実行形式のプログラムを実行する
Line No.1Line No.2 ………改行されずに出力されてしまった
A>
```

図 2-1　リスト2-1の実行結果

あれ？ 期待した出力とは，ちょっと違いますね．2行にわたって出力されないといけないのに，つながって出てきてしまいました．実はC言語で改行する場合は，プログラム中に明確な「改行の印」を入れてやる必要があります．

BASIC のプログラムも実行させてみましょう(**図 2-2**)．

```
RUN⏎
Line No.1 ┐
Line No.2 ┘ きちんと改行している
Ok
```

図 2-2　リスト2-2の実行結果

† CP/M-86 上では「TEST.CMD」，CP/M-80 上では「TEST.COM」というファイルができます

こちらはきちんと目的の文字列を2行に分けて表示しています．BASICでは改行の印を付けなくても，1行書いたらかならず改行するのが暗黙の約束です．したがって反対に改行しない場合には，そのことを明確にするための記号「;」をPRINT文の行末に付けてやる必要があります．

では，**リスト2-1**のプログラムを**リスト2-2**のBASICプログラムと同じように，1行書いた後で改行するように書き直してみます．

```
1: main()
2:     {
3:     printf("Line No.1¥n");
4:     printf("Line No.2");
5:     }
```

改行を示す印

ダブルクォーテーションで囲むのを忘れないこと

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト2-4　改行するように修正したプログラム

リスト2-1のプログラムと違うところは，3行目の文字列の最後に[†]「¥n」という文字が付いているところです．これがC言語で，改行を表す印です．

このプログラムをもう一度コンパイルし直して，実行させてみましょう(**図2-3**)．

```
A>test⏎
Line No.1
Line No.2
A>
```

今度は，きちんと改行された

図2-3　リスト2-4の実行結果

考えたとおりの結果が出てきました．

† ¥記号(キャラクタコード0x5c)は，日本で使われているJISコードとアメリカなどで使われているASCIIコードで違いがあり，ASCIIコードでは"＼"(バックスラッシュ)が割り当てられています．したがって，ASCIIコードに準じたプリンタの出力や国外で作成されたプログラムリストでは¥記号は"＼"に変わっています

■制御文字　―画面に現れない文字―

リスト 2-4の「¥n」のように，画面に表示されずその制御を行う文字を**制御文字**(コントロールキャラクタ)といいます．また，記号「¥」は**エスケープキャラクタ**と呼ばれ，以降に続く1文字を画面にそのまま出力しないようにprintfに指示を与えるための文字です．

「¥n」は，〝次の行の先頭に移れ〟という制御を行いますが，ほかにもいくつか制御文字があります．**表 2-1**にその種類を示します．

制御文字	意　味	機　能
¥n	改行と復帰	次の行の先頭にカーソルを移動
¥t	タブ	次のタブ位置までカーソルを移動
¥b	バックスペース	前に1桁戻る
¥r	復帰(リターン)	同じ行の先頭に戻る

表 2-1　制御文字の種類

C言語では，このような制御文字を表示する文字列のなかに埋め込むことによって，改行やタブなどの画面の制御を行います．

ここで，エスケープキャラクタである「¥」や文字列の範囲を表す「"」(ダブルクォート)には特別な意味があり，このままでは表示できません．これらの記号を表示させたい場合は，

```
printf("……¥" ……");
printf("……¥¥……");
```

というように「¥」の後に文字を重ねます．

ところで，このprintfは「print formatted」または「formatted print-out」の略称で，〝書式指定付き文字出力〟という意味を持っています．BASICのPRINT文と同じく，決まった文字だけでなく数値や変数としての文字なども一定の書式で出力することが可能です．また多彩な出力形式の指定ができるため，その分使い方や書式指定の設定がわかりにくい面もあります．

以下の節では，printfでよく使われる機能を中心に解説します．printfについては，APPENDIX(210ページ)にもくわしくまとめていますので参照されるとよいでしょう．

2.2　変数の画面への出力

この節では，printf を使って変数の値を画面に表示する例題を示します．C 言語で変数を使う場合には，あらかじめ使用宣言をしておく必要があります．変数の宣言方法についてはこの後の節で解説しますので，ここでは〝数値と文字列の変数の出力はこのようにするのだ〞という方法を理解してください．

■数値変数を表示する

まず，数値変数を表示する場合を考えてみます．**リスト 2-5** のプログラムは，変数 a に数値を代入しその値を表示させるものです．

```
1: main() ………プログラムの実行単位の定義(mainという名称は，ほかの名前であってはいけない)
2:     {  ………プログラムの始まり
3:
4:     int    a; ………aという名前の変数の定義
5:
6:     a = 12;  ………変数aに10進数の12を代入
7:
8:     printf("value is %d ¥n", a);
9:     } プログラムの終わり
```

- a) → 表示したい数値の入った変数
- ¥n → 改行の印
- %d → %で始まる文字は,「,」(カンマ)以降の変数の値に置き換わり表示される
- value is → 通常の文字列は，画面にそのまま表示される

MSX-C，LSI C は，1 行目の前に「#include　<stdio.h>」を追加

リスト 2-5　数値変数を表示するプログラム

これをコンパイルして実行させると，**図 2-4** のようになります．

```
A>test↵
value is 12 ………%dのところに変数aの値が置き換わっている

A>
```

図 2-4　リスト2-5の実行結果

リスト 2-5 の 8 行目の表示する文字列には，「%d」という記号が含まれこの位置に「，」(カンマ)以降で指定した変数の内容が置き換わって表示されています．「%d」は，前述の「¥n」と同じように実際に画面に表示されるのではなく，後で指定した変数の表示形式を決めるので，**表現指示のための文字**といいます．また，同様に「%」はエスケープキャラクタです．

この場合「%d」の「d」は「decimal」(10 進数)を意味し，〝変数の値を 10 進数で表示しなさい〟という指示を printf に与えます．このほかにも，**表 2-2** のような表現指示のための文字が決められています．

数　値	
%d	10進数(Decimal)
%x	16進数(heXadecimal)
%o	8進数(Octal)

表 2-2　数値変数を表示するための文字

■ 文字変数を表示する

次に，変数として入った文字を表示する例を考えてみましょう．まず，サンプルとして BASIC プログラムを示し，これと同じように動作する C 言語のプログラムを書いてみます(**リスト 2-6**)．

```
100 A$ = "Jack and Gill"
200 PRINT "*** ";A$;" ***"
300 END
```

リスト 2-6　文字変数を表示するBASICプログラム

このプログラムはC言語で書くと**リスト 2-7** のようになります．

```
1: main()
2:     {
3:     static char a[50] = "Jack and Gill";
4:
5:     printf("*** %s *** ¥n", a);
6:     }
```

staticについては,4章で解説

＊ダブルクォーテーション，カンマ，セミコロンを忘れないように注意

………文字配列a[ ]に"Jack and Gill"という文字列を代入

………a[ ]を画面に表示

%sのところに配列a[ ]の文字列が置き換わる

[実行結果]

```
A>test
*** Jack and Gill ***

A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加
BDS Cは，「3:　char　*a;」，「4:　a = "Jack and Gill";」に変更

リスト 2-7　文字変数を表示するC言語プログラム

3行目には，なんだか見なれない文字の代入の仕方が書いてあります．この行の「static」については，第4章で解説します．また，文字列を扱うのに配列(a[50])を用いていますが，この点は本章の最後でもう一度取り上げます．

ここで見てほしいのは，printf文の書き方です．printfの「%」に続く文字が今度は「s」になっています．これが"文字を表示しろ"という表現指示のための文字です．

■ printfのまとめ

最後にprintfの使い方を**表 2-3**にまとめておきます．

書　式	printf("表示したい文字列", 置き換えたい変数, …);	
目　的	**使い方**	**結　果**
文字を出力する	printf("Jack and Gill");	Jack and Gill
文字変数を出力する	printf("**%s**", a);	a[ ]に"Jack and Gill"が入っていると，**Jack and Gill**
数値変数を出力する	printf("Number is %d", a);	aに12が入っていると, Number is 12
改行を入れる	printf("Line1¥n Line2¥n");	Line1 Line2

表 2-3　printfのまとめ

2.3　数値の扱い

C言語は，変数を使う前にかならず使用宣言をして，その変数の種類を明確にする必要があります．変数の種類はデータ型と呼ばれ，その変数が使用できる数の大きさを表しています．

ここでは，データ型とその宣言方法について述べ，C言語でどのように数値が扱われているのかを見てみましょう．

■変数の種類　―データ型―

C言語には，変数の「型」がいくつか存在します．これらの型は，コンピュータのハードウェアに密着したもので，その処理がしやすいように作られています．コンピュータの最も基本的な処理単位はビット（〝0〟か〝1〟）ですから，変数の型もビット長で分類されています．たとえば，2ビットならば，「00，01，10，11」という4種類の数を表せます．つまり，ビット長は表現できる数の多さを決定することになります．また各ビットの役目を〝最初の20ビットは仮数〟，〝最後の10ビットは指数〟などと区別しておくことによって，私たちが普通に使っている実数（浮動小数点数）なども扱えるわけです．

以下に，C言語で扱える数値のビット長を示します．

- **8ビット幅**　……　256種類の数値が表せる
- **16ビット幅**　……　65,536種類の数値が表せる
- **32ビット幅**　……　4,294,967,296種類の数値が表せる
- **64ビット幅**　……　2^{64}＝およそ18×10^{18}種類の数値が表せる

また，別の分け方をすると，以下のように分類することもできます．

- **符号付き整数**　……　正負の数が表現できる整数
- **符号なし整数**　……　正の数だけが表現できる整数
- **浮動小数点数**　……　正負の数と小数点以下の数やより大きな数を表現できる実数

C言語では，どの程度の大きさの数値を扱うのかによって，変数の型をプログラマ自身が選択する必要があります．この点は，BASICと比べて最もめんどうな点ですが，C言語を使うためには避けて通れないところです．

■変数の宣言方法

実際に変数を使うためには，あらかじめ使用宣言をしなければなりません．〝a〟という変数を使用するための宣言の方法をまとめたのが**表 2-4** です．見てわかるとおり，浮動小数点数は 32 ビット幅のものと 64 ビット幅のものしかなく，整数には 64 ビット幅のものはありません．また，符号なしの場合のみ「unsigned」を付け，通常の宣言では符号付きと見なされます．

表 2-5 には，各データ型で使用できる数の大きさをまとめています．

	符号付き整数	符号なし整数	浮動小数点数
8 ビット幅	char a;	unsigned char a; 3)	なし
16ビット幅	short a; 1)	unsigned short a; 4)	なし
32ビット幅	long a; 2)	unsigned long a; 5)	float a; 6)
64ビット幅	なし	なし	double a; 7)

<各C言語のデータ型の違い>
3)，5)のないC言語
DeSmet C，RUN/C，
MSX-C，LSI C，BDS C
1)を int と表記
BDS C
4)を unsigned と表記
RUN/C,MSX-C,LSI C,BDS C
2)，6)，7)のないC言語
MSX-C，LSI C，BDS C
8)char 型の範囲(0〜255)
DeSmet C，RUN/C，
MSX-C，LSI C，BDS C

表 2-4　C言語で扱えるデータの型

数値の型	数値の範囲	数値の型	数値の範囲
char	−128〜+127 8)	unsigned char	0〜255
short	−32768〜+32767	unsigned short	0〜65535
long	−2147483648〜+2147483647	unsigned long	0〜4294967295
float/double	注) およそ10^{-79}〜10^{75}		

表 2-5　各データ型で表現できる数の大きさ

注)floatとdoubleではデータの精度(仮数部の桁数)が異なる

■int 型　―コンピュータのハードウェアに依存する型―

ここで示したデータ型とは別に，「int」という整数型があります(int とは，integer の略です)．

表 2-4 に示したデータの型は，どのコンピュータのC言語でも同じビット長を表します．しかし，この int 型はそのマシンのハードウェアが最も効率よく処理できる符号付き整数を割り当てることになっています．たとえば 8086CPU やZ80CPU ならば 16 ビットの符号付き整数ですから，short がそ

れに当たり，68000CPUならば32ビット符号付き整数ですからlongということになります．

これから出てくる標準関数(printfなどのこと)は，すべてこのint型を中心に考えて作られていますから，本書でもint型を使ってプログラムを記述していきます．ただし，作ったプログラムをROMに焼き付け，ハードウェアの制御などに使う場合は，intを使用しない方が扱う数値が明確になり，よい結果を生むことが多いようです．

なお，本書ではint型は16ビット符号付き整数として扱います．

■各種のデータ型を使ったプログラム

さて，ひととおりC言語で扱うことのできる数値を見てきました．ここで，これらを使ったプログラムを前述のプログラム例を少々書き換えて試してみることにしましょう(**リスト2-8**)．

```
 1: /*   ………コメントの始まり
 2:      This is sample program for any formed value.
 3:       ( This line is comment-area. Compiler will ignore this area. )
 4: */   ………コメントの終わり
 5:
 6: main()
 7:      {
 8:                long  a;  /* a is 32bit-width signed value    */
 9:      unsigned int  b;  /* b is 16bit-width unsigned value */
10:                char  c;  /* c is  8bit-width signed value    */
11:
12:      a = -120000;      /* a is -120000 in decimal        */
13:      b = 0x3FFF;………16進数 /* b is 0x3FFF  in hexiadecimal */
14:      c = 015;   ……… 8進数 /* c is 015     in octal          */
15:
16:      printf("value a is %d ¥n", a);   ………10進数で表示
17:           /* output a in decimal      */
18:      printf("value b is %x ¥n", b);   ………16進数で表示
19:           /* output b in hexadecimal  */
20:      printf("value c is %o ¥n", c);   ……… 8進数で表示
21:           /* output c in octal        */
22:      }
```

日本語の使える処理系では日本語でコメントを書いておくとより見やすい

[実行結果]

```
A>test↵
value a is -120000
value b is 3fff
value c is 15
A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加
MSX-C，LSI C，BDS Cは，long型を使用できないので，「8: int a;」，「12: a = −12000;」に変更
BDS Cは，「9: unsigned b;」に変更
RUN/Cは，「9: unsigned b;」，「12: a = −120000l;」に変更
PM-MWC，HI-TECH C，DeSmet C，Lattice C，RUN/Cは，「16: printf("value a is %ld ¥n", a);」に変更

リスト2-8 各種のデータ型を使ったサンプルプログラム

このプログラムには，これまでに説明しなかったものをいくつか使っています．

まず，プログラム中には**コメント**が書かれています．C言語のコメントは，「/*」で始まり「*/」で終わります．コメントが複数行にわたるときでも，最初の行に「/*」を，終わりの行に「*/」を付ければ，その間はすべてコメントということになり，コンパイラは読みとばしてくれます．

コメントは,後でプログラムを読み返すのが楽なようにたくさん書いておく癖をつけておきましょう．

8行目～10行目は，このプログラムで使用する変数のデータ型を宣言しています．ここでは，3つの変数にそれぞれ違ったデータ型を割り当てています．

12行目～14行目では，宣言された変数に数値の代入を行っています．しかし，なんだか代入されている数値が不思議な形をしています．変数aの値を入れているところは10進数そのものですが，変数bとcの値はそうではありません．

13行目でbに代入する数値は，"0x3FFF"となっています．C言語では数値定数を表現するとき，その頭に0x(0(ゼロ)と小文字のx)があると，その数値を16進数として解釈します．すなわち，この値は16進表現で3FFF(10進数で16383)ということになります．また，変数bはunsigned(符号なし)で宣言しているため，もしここに入る値の一番上のビットが1(たとえば0xF000)であっても，この数は10進数で表現すると負の数にはなりません．反対に符号付きで宣言してあれば，同じ数値の表現でも負の数ということになってしまいます．**図2-5**に符号付きと符号なし整数の関係を示しておきます．

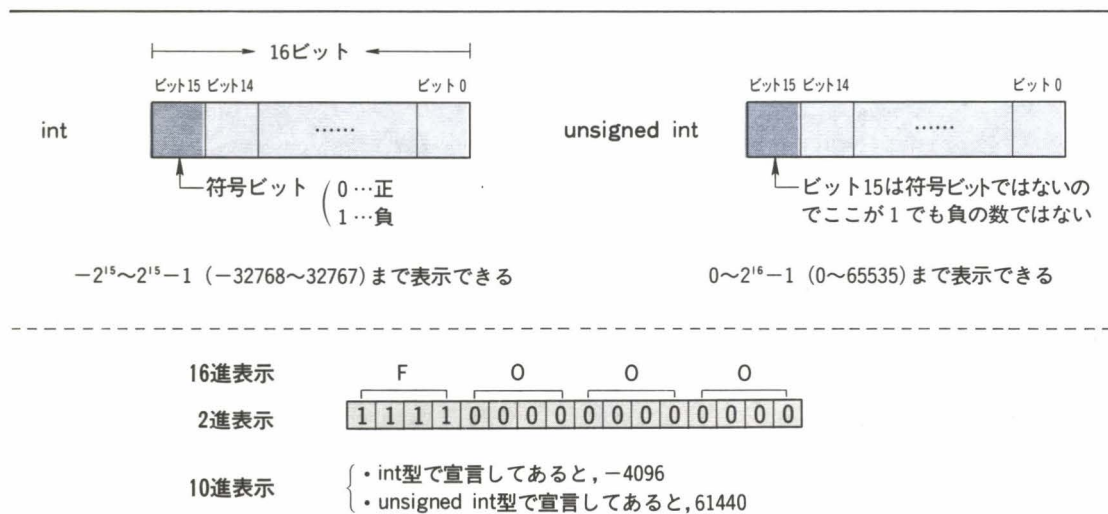



図 2-5　符号付きと符号なし整数

このことは演算のときや画面に数値を出力させる場合などで，問題となります．画面への出力では結果が目に見えるので間違いもすぐにわかることが多いのですが，演算の途中などに間違えて宣言した変数が入っていると，結果がでたらめになってしまいます(この手のバグ取りは，かなり時間がかかりますし根気のいる作業です)．おまけにC言語はオーバーフロー(桁あふれ)のチェックはいっさい行わないので，このようなエラーはまかり間違うと実行時の暴走の原因にもなりかねません．

変数の宣言のときは，その変数がとりうる最大，最小の値をしっかりと把握してから，型を決めましょう．

次に14行目で変数cの代入をしている行ですが，〝なぜこれが8進数なのだろう〞と思われた人も多いでしょう．C言語では数値の頭に0が付くと，その数は8進数と見なすのです．したがって，このような数値の表現はすべて8進数となります．ということは，〝015〞と〝15〞と書く場合ではまったく別の数になってしまいますので注意してください．

種　類	2340という数値の表記	表記の方法
10進数	2340	
8進数	04444	先頭に0を付ける
16進数	0x924	先頭に0xを付ける

表 2-6　8進数と16進数の表記

コメントに泣く

よく初心者が(実は初心者に限らないのですが)，計らずもしてしまう間違いとしてよく知られているのは以下のようなケースです．

プログラムの最初に「/*」を書いておいたのを忘れて，そのまま長いプログラムを書き終わりコンパイルをします．コンパイルはエラーなしで無事に終了し，「なんと私はすばらしいプログラマなのだろう！ こんな長いプログラムを最初からまったくのノーエラーで通してしまうなんて…」と，震える右手を左手で押さえ，厳かにプログラムを動かしてみる……．と，プログラムは動かずただ空しくプログラムの動作終了を表すプロンプトが返ってくるばかりである……．

要するに，コメントの終わりで「*/」を書いておかなかったために，コンパイラはプログラムすべてをコメントとみなし，コンパイルをいっさいせずに終了してしまったのでした．確かにこれではエラーが出るはずはありません．みなさん，十分気をつけてください．

2.4　文字列と配列

C言語での文字列の扱いは，数値の場合と同様にBASICとは扱いが異なります．BASICでは1つの文字変数に文字列を代入することができますが，C言語では1つの文字変数に1文字しか入れることができません．また，C言語でいう文字列は他の言語とは少し違う点があります．

この節では，文字列の中身を見ることによってC言語での文字列の扱いを考えてみます．

■C言語の文字変数

BASICでは1つの文字変数を用意すると，**リスト 2-9**のようにその変数に任意の文字数(通常は255文字まで)の文字列を入れることができます．

```
100 A$ = "A"   : PRINT A$
200 A$ = "AB"  : PRINT A$
300 A$ = "ABC" : PRINT A$
400 END
```

リスト 2-9　BASICでの文字列の扱い

これに対してC言語は，1文字を8ビットのデータ型である「char」で宣言します(基本的に16ビットの漢字コードなどは扱えません)．つまりBASICのように，文字変数の大きさは変化せず1つの変数には8ビットで表せる1文字しか入りません．そして文字列は，char型の変数の配列で表します．このため文字列を扱う場合には，あらかじめ使用する文字数に応じた配列を確保しておく必要があります．

たとえば，**リスト 2-10**の例を見てみましょう．

```
1: main()
2:      {
3:      char      string[2];………文字変数の配列であるstring[2]を宣言
4:                                  このとき使用できる配列は (string[0]) である
                                                             (string[1])
5:      string[0] = 'A'; } 使用できる
6:      string[1] = 'B'; }
7:      string[2] = 'C';………string[2]は使用できない
8:           :
9:           :
                 1文字の場合は，シングルクォーテーションで囲む
```

リスト 2-10　C言語での文字列1

この例では3行目で「string」という名前の配列を，〝ABC〟という文字列を入れるために宣言しています．しかし，この宣言文では，2文字分しか宣言していないため，7行目の代入文で入れようとしている文字があふれてしまいます．このような場合でも，多くのC言語のコンパイラはエラーメッセージを出しませんので注意してください．配列の要素の数は，次に示す**リスト 2-11**のように大きくしておかなければなりません．

```
1: main()
2:     {
3:     char      string[5]; ………配列の数を大きくする
4:
5:     string[0] = 'A';
6:     string[1] = 'B';
7:     string[2] = 'C';  ………string[2]も使用できる
8:          ⋮
9:          ⋮
```

リスト 2-11　C言語での文字列 2

■配列の考え方

さて**リスト 2-11**で，無事に「string」という配列変数に〝ABC〟という文字が入ったとしましょう．しかし，これだけではC言語の文字列にはならないのです．C言語で扱う文字列は，**Cストリング**と呼ばれ，その他の言語で扱う文字列とはっきり区別しています．

ではC言語で扱う文字列とは，いったいどんな文字列なのでしょうか．せっかく文字列が配列として扱われていますから，**リスト 2-12**のようなプログラムを作ってCストリングのなかをのぞいてみましょう．

```
 1: main()
 2:     {
 3:     static char string[5] = "ABC"; ………文字変数の配列であるstringに
 4:                                          "ABC"を代入する
 5:     printf("No.0 is %x¥n", string[0]); ………1文字目を16進数で表示
 6:     printf("No.1 is %x¥n", string[1]); ………2文字目を16進数で表示
 7:     printf("No.2 is %x¥n", string[2]); ………3文字目を16進数で表示
 8:     printf("No.3 is %x¥n", string[3]); ………4文字目を16進数で表示
 9:     printf("No.4 is %x¥n", string[4]); ………5文字目を16進数で表示
10:     }
```

「表現指示のための文字」を用いることによって文字として入力されたものを16進数や10進数に変換して表示することができる

MSX-C，LSI Cは，1行目の前に「#include　<stdio.h>」を追加．5行目から9行目のstringの前に「(int)」を追加
BDS Cは，「3:　char　*string;」，「4:　string = "ABC";」に変更

リスト 2-12　文字列の中身

ここで，〝あれ？〟と思われた方もいるでしょう．〝5文字分の宣言をしている〟と言っておきながら，後のprintf文で使用している「string」の添字は0から4までしか使用していません．

実はC言語で配列を宣言する場合，宣言される数は，確保すべき領域の〝個数〟という決まりになっています．またそれを使う場合には，添字を0から使うように決まっています．ですから，5で宣言すれば使える添字は0から4ですし，100で宣言すれば使える添字は0から99ということになります．

BASICの配列に慣れた人には，ずいぶんややこしい気がするかもしれませんが，配列の最初の数が1でも0でも宣言の仕方で変わってしまうことはC言語ではいっさいないので，慣れてしまえばかえってわかりやすいでしょう．

■Cストリング　―C言語で扱う文字列―

さて，このプログラムをコンパイルして，実行させてみましょう．実行結果は**図2-6**のとおりです．

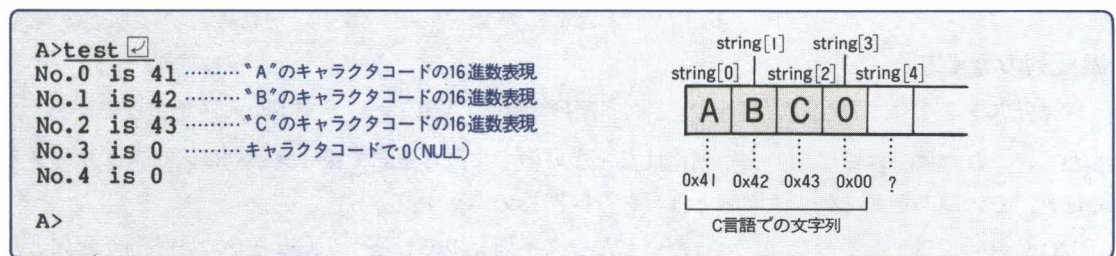



図2-6　リスト2-12の実行結果

最初の3文字分は確かに入れたはずの文字のコード(209ページ参照)が入っています．その次の〝0〟は，いったい何でしょう？

この0こそが，C言語の文字列であることの証しといってもよい，重要な存在なのです．C言語は，**文字列の終わりをかならずこの0でくくります**(0を**ヌル文字**といい，これも1つのキャラクタです)．そしてこの0を見ることによって，たとえばprintfはその文字列の終わりを見きわめて，ここから先の領域は画面に表示しないようにするのです．したがって，もしこの0がない文字列がprintfに渡ると，文字列の終わりがわからなくなり，プログラムは暴走してしまいます．

試しにこのプログラムの一部を書き換えて，12行目のstring[3]に0以外の数値を入れてから，printfで変数stringを画面に表示させるプログラムを作ってみてください．おそらく暴走するか，そ

れに近い状態になるはずです．もし，無事にプログラムが動いた場合は，運よく〝ABC〃の文字のすぐ先に0があったのだと考えられます．また逆に，string[1]に0を代入して表示させてみましょう．たとえそれ以降に文字があっても，printfは0の前まで(すなわち41)しか表示しません．

つまり，前述の**リスト 2-11**は7行目の後に，

```
string[3] = 0;
```

の1行を加えてやらなければ，C言語の文字列にはならないのです．

■文字配列を扱う上での注意

C言語で文字列を扱う場合には，それを構成する文字以外にもう1つ〝0〃という文字を必要とします．このことは，文字列を入れるために配列を宣言する場合，かならず〝実際に必要とする文字数よりも，余分に1バイトを見込んで宣言する必要がある〃ということです．

メモリがかなりの低価格で市場に出回っている近ごろでも，とにかくエリアを節約したくなる人は，C言語を確実に動かすおまじないの1つとして**配列の宣言は多めにしておくということを**，覚えておいて損はありません．

C言語における文字列をまとめると，**図 2-7**のようになります．

Cストリング（C言語での文字列の単位）

文字列の始め　　文字列の終わり

c		i	s		b	e	a	u	t	i	f	u	l	!	¥n	00

改行コードなどの制御文字も文字に数えられる

ヌル文字(NULL)

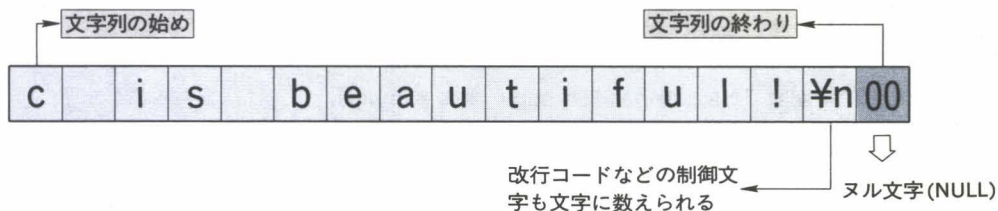



図 2-7　Cストリングのまとめ

【この章のポイント】

1．C言語は，かならず「main( ) { }」という入れ物がいる
2．C言語は，実行の単位ごとに「;」(セミコロン)で区切る必要がある
3．数値や文字の画面への出力はprintfを使う
4．C言語で扱える数値の種類(int，short，long，float，double型)
5．符号付きと符号なしのデータの扱い
6．文字列は，char型の変数の配列である
7．文字列の宣言は，＜必要な文字数＋1＞以上とする
8．配列のなかのデータを見るには，0から，＜宣言した個数－1＞までの添字を使う

[練習問題]

1．次の実行結果が得られるプログラムをC言語で書きなさい(数値の部分を文字列で出してはいけません)

```
A>test↵

Number is 3f21 in hexadecimal.
Number is 16161 in decimal.
Number is 37441 in octal.

A>
```

＊注　これらは表現が違うだけで，すべて同じ大きさの値です

2．次のprintf文の間違いを捜しなさい

```
printf("＊＊＊ This line has %d bugs. ＊＊＊, num)
```

3．次の「test」という配列の内部を図で示しなさい

```
static char    test[20]="Time is 10:00" ;
```

4．以下の行を含むプログラムをコンパイルし，その実行結果を確かめなさい

```
printf("This is test /＊ comment ＊/ for comments");
```

(解答は217ページ)

第3章
基本的なC言語のプログラミング

この章では，文字や数値のキーボードからの入力，計算のための演算子，そしてループや条件判断を行う制御構造について学んでいきます．

C言語は，よく「構造化言語」と呼ばれます．構造化という言葉は，英語では「structured」となるわけで，〝1つのまとまりを持って作られたもの〟という意味を持っています．つまりC言語は，あるまとまりを単位として構成される言語ということになります．そしてこのまとまりを構成し，プログラムの骨組みを作る最も重要な要素が「制御構造」なのです．しかし，制御構造についての基本的な考え方というのは，どのプログラミング言語でもそれほど変わることはありません．要は，C言語のスタイルに慣れてしまうことです．

またC言語の演算子は，他の言語に比べると実に多くの種類があります．しかし，そのほとんどは基本的ないくつかの演算子の組み合わせでできており，その組み合わせ方もごく単純な規則があるに過ぎません．これらの演算子は，使い慣れるとずいぶん便利なものです．それだけでなく，この演算子を使うことによってプログラムの実行速度を向上させることができるのです．最初のうちは，その演算子の数の多さにとまどうことがあるかもしれません．実際にプログラミングに使用する演算子はごくわずかなので，はじめは必要なものだけを覚えればよいでしょう．まずは軽い気持ちで使ってみるということが大切です．

3.1 文字や数値の入力

第2章では，文字や数値を画面に出力する例題を取り上げました．またC言語でのプログラミングについても，おおよその感じはつかめたことでしょう．文字や数値の出力ができるようになると，次に学ぶことは，なんといっても入力です．ここでは，printfと同じ形式で数値や文字列を入力できる「scanf」について見ていきます．

■ scanfを使った数値と文字列の入力

scanfとは「scan formatted」のことで，入力された文字や数値を指定された形式で変数に取り込む働きをします．つまりscanfは，printfとちょうど逆の働きをするわけです．printfでは数値や文字列を画面に出力しましたが，scanfはキーボードから入力されたデータをC言語のプログラム中の変数に取り込みます．

まずは，第2章の復習もかねて，printfを使った**リスト 3-1**のような例題を考えてみます．

```
 1: main()
 2:      {
 3:      static char a[25] = "This is test line.";
 4:      int    b;
 5:
 6:      b = 0x300;
 7:
 8:      printf(" **** Strings is : %s ****¥n", a);
 9:      printf(" **** Value   is : %x ****¥n", b);
10:      }
```


[実行結果]

```
A>test
 **** Strings is : This is test line. ****
 **** Value   is : 300 ****
A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio . h>」を追加
BDS Cは，「3: char *a; a = "This is test line.";」に変更

リスト 3-1 printfを使った画面への出力

リスト 3-1の例では，変数aとbに文字列と数値を代入し，それをprintfで出力しています．これに手を加えて，キーボードからこれらの値を入力できるように変更します(**リスト 3-2**)．

```
 1: main()
 2:     {
 3:     char    a[25];……… 文字列を入れる配列を宣言
 4:     int     b;    ……… int型の数値を入れる変数を宣言
 5:                ┌─→文字列として変数に取り込む
 6:     scanf("%s", a);…………aという配列にキーボードから文字列を入れる
 7:     scanf("%x", &b);…………bという変数にキーボードから16進数を入れる(&に注意!!)
 8:                └─→16進数として変数に取り込む
 9:     printf(" **** Strings is : %s ****¥n", a); ┐
10:     printf(" **** Value   is : %x ****¥n", b); ┘変数aとbの表示
11:     }
```

[実行結果]

```
A>test
test-line ………"test-line"と入力する ┐
3FC ……………………"3FC"と入力する      ┘BASICのINPUT文と違い,"?"などの記号は出てこない
 **** Strings is : test-line ****
 **** Value   is : 3fc ****
A>
```

MSX-C, LSI C は, 1行目の前に「#include　<stdio.h>」を追加

リスト 3-2　scanfを使ったキーボードからの入力

あれ？ またまた初めての記号がありますね．はじめから，順を追って見ていきましょう．

最初の4行と最後の3行は省略します．よくわからない人は，前の章に戻ってもう一度おさらいをしてください．

6行目が問題のscanfです．まず実行結果を先に見ると，BASICのINPUT文と違って〝?〟という入力を促す記号が出てこないことに気がつきます．つまり，scanfが実行されると何も表示せずにキーボードからの入力を待っているのです．そして値が入力されると，指定された形式に合わせて変数に代入が行われます．この場合は，前章でやった表現指示のための文字「%s」が付いているので，入力された値を文字列として取り込むことになります．

7行目のscanfは，変数bの前になぜか「&」という記号が付いています．これを忘れるとこのプログラムは正常に動作しません．この「&」とはいったい何なのでしょう？

この「&」は，〝変数bの存在する場所〟を表すのです．これを「ポインタ」と言います．ポインタについては，次章で取り上げますので，ここではscanfで数値を扱うために必要な約束事の1つであると考えてください．

scanfとprintfは，その書式がほとんど同じです．ただし，scanfの場合は次のように記述することはできません．

```
scanf("INPUT STRING : %s", a);
      └────ここに文字列を書くことはできない
```

■ BASICと同じように入力できるプログラムを作る

さて，scanfはprintfと同じように表現指示の文字を用いて文字列や数値の入力ができますが，BASICのINPUT文のように文字列を書くことができないので不便です．そこで，INPUT文と同じ感覚でデータ入力ができるプログラムを作ってみます．まず，**リスト 3-3** にBASICのプログラムを示します．

```
100 INPUT " INPUT DATA : ";DX
110 PRINT " >> ";DX
120 END

[実行結果]
RUN↵
 INPUT DATA : 150↵
 >> 150
```

リスト 3-3　BASICによる入力文の記述

同じことをC言語で記述すると**リスト 3-4** のようになります．

```
1: main()
2:      {
3:      int   dx;
4:
5:      printf(" INPUT DATA : ");………入力を促す文字列の表示(¥nを入れると次の行にカーソルが移動してしまうので注意)
6:      scanf("%d", &dx); ………数値の入力(&に注意)
7:      printf(" >> %d¥n", dx); ………入力された数値を表示する
8:      }

[実行結果]
A>test↵
 INPUT DATA : 150↵
 >> 150
A>
         入力を促す文字列

MSX-C, LSI Cは，1行目の前に「#include <stdio.h>」を追加
```

リスト 3-4　リスト3-3と同じ働きをするC言語のプログラム

このように，あらかじめprintfで入力を促す文字列を出してやれば，INPUT文と同じように使うことができます．scanfはprintfと組みにして使うと覚えておけばよいでしょう．

リスト 3-2 と **3-4** で示した例題は，データ型としてint型とchar型を，表現指示のための文字として「%s」，「%x」，「%d」を取り上げましたが，その他のデータ型や表現指示の文字についても，自分でプログラムを書き換えて確認してください．

■scanfを使う上での注意

scanfは実際はファイルからの入力を想定しているので，多くのC言語では〝何も入れずにリターンキーを押すと値が入力されない″といったことがあります．また，文字列の途中でスペースを入れると，スペースの前までしか変数に取り込んでくれません．つまり，scanfの入力で，

```
13　15↵
```

と数値を入れると「13」のみが入ることになります．

scanfで入力する場合は以上の点を，十分注意してください．先の**リスト 3-2** で正しく入力されない例を示しておきます(**図 3-1**)．

```
A>test↵
↵……………リターンキーのみが入力されると，無視される
today↵ ┐
100 ↵  ┘………正しい入力をすると結果が表示される
 **** Strings is today ****
 **** Value   is 100 ****

A>test↵
this is↵………ブランク(空白)を含んだ文章を入れると，そこで文字列の終わりと認識する
 **** Strings is this ****
 **** Value   is 3860 ****………〝is″を16進数で表した数値

A>
```

図 3-1　リスト3-2の誤った入力例

■scanfのまとめ

前章で学んだprintfと本節のscanfは，C言語で入出力を行う場合の中心となる関数です(C言語ではコマンドといわずに関数といいます)．C言語にはこのほかにも，キーボードからの入力や画面への出力のためのたくさんの関数があります．しかし，最初はこの2つの関数を知っていれば，ほかに画面やキーボードの入出力の関数を知っている必要はありません．BASICでいえば，INPUT文と

PRINT 文に当たるわけですから，これをもとに少しずつ知識を増やしていけばよいのです．

最後に，scanf の使い方を**図 3-2** にまとめておきます．

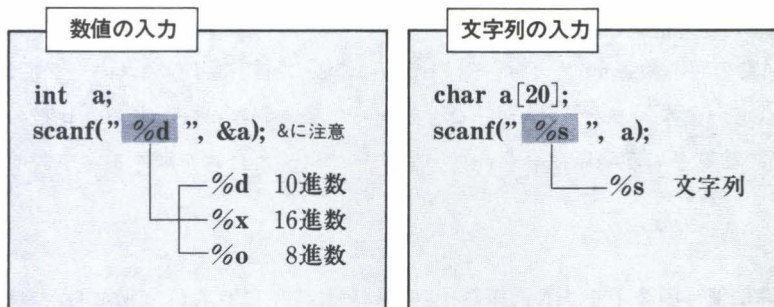



図 3-2　scanfのまとめ

scanfのバグに注意

scanf 関数は，これまで示したように手軽に使うことができて便利なのですが，どこのＣ言語でもバグの多い標準関数として有名です．また，文字列を変数に入力するときに，あらかじめ宣言しておいた変数の大きさを越える値を入力されてしまうといったことも起こります．自分だけでなく他の人も使うプログラムを組む場合には，とくに値の入力について気をつかわなければなりません．
そこで，通常アプリケーションを記述する際には，

```
scanf("%d", &i);
```

とするところを，たとえば

```
gets(is);       …… 1行を is[ ]という配列に入れる標準関数
i = atoi(is);   …… is[ ]のなかの数値を表す文字列を，整数に変換する標準関数
```

などと書くことが多いようです(これでも，上に書いた不都合が起こる可能性はありますが…)．

プログラムが組めるようになったら，まずは自分で数値や文字の入力用の関数を別に作ってみるとよいでしょう．

3.2 計算をするには ―演算子の扱い―

計算は，プログラムの基礎となる課題の１つです．ここでは，計算をするための演算子の扱いについて学んでいくことにしましょう．

C言語には，通常の四則演算(＋, －, ×, ÷)のほかに，他の言語にはないユニークで便利な演算子がたくさん用意されています．また，BASICなどと違って条件判断に使われる「比較のための演算子」と，「計算のための演算子」が明確に分かれています．では，これらを順を追って説明します．

■算術演算子

ごく普通の計算に使う演算子を「算術演算子」といいます．これらは，私たちが普段使っている四則演算とほとんど同じものです．書式やかっこの使い方も，通常の計算方法と違うところはありません．**表 3-1** にこれらの種類を示します．

演算	記号	例	意　味
加算	＋	a＋b	aにbを加える
減算	－	a－b	aからbを引く
乗算	＊	a＊b	aとbを掛ける
除算	／	a／b	aをbで割る(注)
剰余算	%	a%b	aをbで割った余り

(注)整数どうしの除算で余りが出る場合は，答えは小数点以下を切り捨てた値となります

表 3-1　算術演算子の種類

このあたりは，あまりに〝当り前〟なので眠くなる方もいるでしょう．眠くなった場合は適当にとばして，次の項目をご覧になってください．

このあとに続く「代入演算子」や「インクリメント／デクリメント演算子」は，ほとんどが算術演算子で記述できるものばかりです．つまり，算術演算子を知っていれば，ほとんどの計算ができることになります．

算術演算子を使ったプログラムを**リスト 3-5** に示します．

```
 1: main()
 2:      {
 3:      int    a, b; ………変数の宣言文は，カンマで区切って書くこともできる
 4:
 5:      a = 3;
 6:      b = 2;
 7:
 8:      printf(" a + b = %d¥n", a + b);
 9:      printf(" a - b = %d¥n", a - b);
10:      printf(" a * b = %d¥n", a * b);   printf中の変数のところには，式をそのまま書いてもよい
11:      printf(" a / b = %d¥n", a / b);
12:      printf(" a %% b = %d¥n", a % b);
13:      }
```

"¥"と同じように"%"を表示させる場合も"%%"と書く（%dなどの指示のための文字でないことを示すため）

[実行結果]

```
A>test
 a + b = 5
 a - b = 1
 a * b = 6
 a / b = 1 ………除算は切り捨て
 a % b = 1
A>
```

MSX-C，LSI C は，1行目の前に「#include <stdio.h>」を追加

リスト 3-5　算術演算子のサンプルプログラム

■代入演算子

算術演算子を拡張し，より少ない文字数で決まった代入計算をできるようにしたのが「代入演算子」です．この演算子を使うことによって，表現が簡略化されるためプログラミング作業の能率も高くなります．また，コンパイラがコンピュータの命令に変換するときも，より少ない量の命令語に変換されるので，コンパイルの効率も上がり実行速度も速くなります．

もっとも，〝けっこう種類があるので全部覚えるのは大変だ〟という声も聞かれます．前の項でも書いたように，これらは算術演算子で記述することが可能です．慣れないうちは，算術演算子で書いておいてもよいでしょう．

表 3-2 にこれらの演算子の種類と使用例を掲げます．

演算子	使用例	意　味	算術演算子での記述
＝	a＝b;	aにbを代入する	
＋＝	a＋＝b;	aにbを加えた値をaに代入する	a＝a＋b;
−＝	a−＝b;	aからbを引いた値をaに代入する	a＝a−b;
＊＝	a＊＝b;	aにbを乗じた値をaに代入する	a＝a＊b;
／＝	a／＝b;	aをbで割った値をaに代入する	a＝a／b;
%＝	a%＝b;	aをbで割った余りをaに代入する	a＝a%b;

表 3-2　代入演算子の種類

ここに掲げた代入演算子は，C言語で使えるものの半分ぐらいです．あとは少し特殊なものなので，ここでは取り上げません．

表 3-2 に書かれた算術演算子に比べると，代入演算子では変数名を書く回数が減っています．そのために間違いが少なくなり，何よりもこのような表現にすることによって，〝2つの変数の間で何が行われたか〞が一目でわかります．すなわち，この演算子を覚えていれば，計算して代入する演算がよりシンプルに書けてしまうのです．

■インクリメント演算子とデクリメント演算子

インクリメントとは，1を加えるということです．またデクリメントとは，1を引くということです．この演算子は，変数に1を加えたり，引いたりするためだけに用意されています．

表 3-3 にその演算子を示します．

演算子	使用例	意　味	代入演算子での記述	算術演算子での記述
＋＋	a＋＋;または＋＋a;	aに1を加える	a＋＝1;	a＝a＋1;
−−	a−−;または−−a;	aから1を引く	a−＝1;	a＝a−1;

表 3-3　インクリメント／デクリメント演算子

これらの演算子は，他の演算子と組み合わせて使用することができます．この際に注意しなければならないことは，「＋＋」と「－－」がある位置によって演算の方法が異なることです．たとえば，

b ＝ ＋＋a；……… aに1を加えてから，その値をbに代入する

b ＝ a＋＋；……… aをbに代入してから，aに1を加える

となり，場合によって計算結果が異なってくることがあるので注意が必要です．

この演算子は，ループ中のカウンタなどとしてかなり頻繁に使われます．また，ほかにも便利な使い方ができるので，覚えておいて損のない演算子の1つです．

代入演算子とインクリメント／デクリメント演算子の実際の使用例を，**リスト 3-6** に掲げておきます．

```
 1: main()
 2:     {
 3:     int   i, j;
 4:
 5:     printf("input<i> =: ");  }iの値を入力する
 6:     scanf("%d", &i);         }
 7:
 8:     printf("input<j> =: ");  }jの値を入力する
 9:     scanf("%d", &j);         }
10:
11:     printf("¥n Increment<i> => %d¥n", ++i);……iの値を1だけ増やしてから表示
12:     printf(" Decrement<j> => %d¥n", --j); ………jの値を1だけ減らしてから表示
13:
14:     i *= j; ………i=i×jを計算する
15:
16:     printf(" i * j = %d¥n", i); ………計算したiの値を表示
17:     }
```

[実行結果]

```
A>test↵
input<i> =: 5 ↵  }iとjに値を入力
input<j> =: 7 ↵  }

 Increment<i> => 6 ………インクリメントされたiを表示
 Decrement<j> => 6 ………デクリメントされたjを表示
 i * j = 36
A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト 3-6 代入演算子とインクリメント／デクリメント演算子のサンプルプログラム

■比較演算子

if 文などの制御文で，比較に使われる演算子が「比較演算子」です．これも BASIC などの比較演算子とほぼ同じものですが，一部違うところがあります．

たとえば，〝等しい〟を表す「＝」はC言語では「＝＝」と，「＝」を２つ並べます．また，〝等しくない〟ことを表す「＜＞」はC言語では「！＝」となります．英語で等しくないことを「not equal」といいますが，この「！＝」はなんだかその表現を意識しているようです．

比較演算子を使った式の結果は if 文などの論理演算で使用されるため，かならず１か０になります．すなわち，

真(true)の場合　……　演算結果は１
偽(false)の場合　……　演算結果は０

となっています．**表 3-4** に比較演算子を示します．

演算子	使用例	意　味
＜	a＜b	aはbより小さい
＞	a＞b	aはbより大きい
＜＝	a＜＝b	aはbより小さいか等しい
＞＝	a＞＝b	aはbより大きいか等しい
＝＝	a＝＝b	aとbは等しい
！＝	a！＝b	aとbは等しくない

（注）式の値は（正しい場合(真)　：１／正しくない場合(偽)：０）となる

表 3-4　比較演算子の種類

とくに，〝等しい〟を表す「＝＝」は，〝代入〟を表す「＝」と明確に区別されています．BASIC に慣れ親しんできた人は，この２つの記号を取り違えてしまうことがよくあります．かくいう私もその一人で，「＝＝」と「＝」を取り違えるという間違いは一度ではありません．本書でC言語を勉強される方は，くれぐれも注意してください．

比較演算子を使ったサンプルプログラムを**リスト 3-7** に示します．

```
 1: main()
 2:      {
 3:      int   i, j, x, y, z;
 4:
 5:      i = 7; j = 5;………変数iとjに値を代入する(このように並べて記述することも可能)
 6:
 7:      x = i > j;  )
 8:      y = i < j;  } 比較演算の結果を変数 x, y, z に代入
 9:      z = i == j; )
10:
11:      printf(" i > j ==> %d¥n", x); )
12:      printf(" i < j ==> %d¥n", y); } 比較演算の値を表示する
13:      printf(" i == j => %d¥n", z); )
14:      }
```

[実行結果]

```
A>test⏎
 i > j ==> 1 ……… 真の場合の演算結果は1
 i < j ==> 0 )
 i == j => 0 } …… 偽の場合の演算結果は0
A>
```

MSX-C, LSI C は, 1行目の前に「#include <stdio.h>」を追加

リスト 3-7 比較演算子のサンプルプログラム

■論理演算子

プログラム中で, if 文などの論理判断を必要とするとき, 複数の条件間で論理演算を行うのに使われるのが「論理演算子」です. 具体的には, 条件のあいだで AND(かつ)とか, OR(または)とか, NOT(否定)といった演算を行うものです.

表 3-5 にそれらを示します.

演算子	使用例	意　味
&&	a&&b	論理積(AND)
¦¦	a¦¦b	論理和(OR)
!	!a	否定(NOT)

表 3-5 論理演算子の種類

論理演算子は，他の言語とは少し違っています．とくにANDとORを表す「&&」，「¦¦」は，2度記号を並べる点に注意してください．

リスト 3-8 に論理演算子を使ったプログラムを示します．

```
 1: main()
 2:     {
 3:     int   i, j, x, y;
 4:
 5:     i = 7; j = 5;
 6:
 7:     x = i > j;
 8:     y = i < j;
 9:
10:     printf(" x and (not y) ==> %d¥n", (x && (!y)));
11:     printf(" (not x) or y  ==> %d¥n", ((!x) || y));
12:     }
```


[実行結果]

```
A>test
 x and (not y) ==> 1
 (not x) or y  ==> 0
A>
```


MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト 3-8　論理演算子のサンプルプログラム

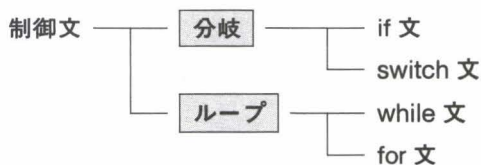
■ その他の演算子

このほかにもC言語には，**ビット演算子**や**三項演算子**などのさまざまな演算子がありますが，本書では取り上げません．ここで紹介した演算子を覚えておけば，C言語での数値や論理の計算のほとんどが可能になります．

3.3　処理の流れの制御 －C言語の制御構造－

処理の流れを記述することは，プログラミングを行う上での最も重要な要素の1つです．C言語には，処理の流れを制御し，すっきりとまとまったプログラムを書くために必要な構文(制御構造)が備わっています．このようにまとまりを持ったプログラムを書くことを**構造化**と呼びますが，ここでは制御構造を学ぶことで，その一端を見ていくことにします．

■制御文の種類

制御文には，処理を分岐させるものとループ(繰り返し)を形成するものの2種類があります．分岐の制御には，おなじみの「if文」のほか，2つ以上の処理に流れを分岐する「switch文」が用意されています．ループの制御には，「while文」と複雑なループを簡潔に記述できる「for文」があります．

C言語には，BASICなど他のプログラミング言語でよく使う「goto文」もありますが，ほとんど使用しません．それは，"{ }"で実行文を囲むことによって，制御文が対象とする処理範囲(実行単位)を明確にすることができるからです．C言語ではgoto文を使わないために，複雑な制御構造も非常に見やすく記述できます．

ほかにも制御文が用意されていますがこの4つを知っていれば，ほとんどの処理の流れを制御することが可能です．

■if文

さて，最初に登場するのは「if文」です．分岐の制御だけは，どの言語でもかならず「if」という単語を使っています．C言語でもそれは同じです．

リスト3-9のプログラムを見てみましょう．ここでは前節で学んだ比較演算子がif文の条件として使われています．このプログラムは，scanfで入れた2つの値をif文で比較して，その結果をprintfで表示させるものです．ここでは「if文」と一言で表現していますが，**リスト3-9**のように実際は「else」

も含んだ文になることが多く，正確には「if～else文」といった方が当たっているかもしれません．

さて，プログラムをじっくり見てみましょう．とくに，細かい点(かっこの使い方など)に注意してください．

```
 1: main()
 2:     {
 3:     int   a, b;
 4:
 5:     printf(" First value  : ");
 6:     scanf("%d", &a);………最初の値の入力(&に注意)
 7:
 8:     printf(" Second value : ");
 9:     scanf("%d", &b);………2番目の値の入力(&に注意)
10:
11:     if(a == b)………aとbが等しいかどうかの条件判断(セミコロン";"は付けない)
12:         {
13:         printf("¥n First == Second ¥n");   a==bが真のときの実行単位
14:         }
15:     else ………セミコロン";"は付けない
16:         {
17:         printf("¥n First != Second ¥n");   a==bが偽のときの実行単位
18:         }
19:     }
```

［実行結果］

```
A>test↵
 First value  : 7↵
 Second value : 3↵

 First != Second ………最初の値と2番目の値は等しくない

A>test↵
 First value  : 11↵
 Second value : 11↵

 First == Second ………最初の値と2番目の値は等しい

A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト 3-9　if文のサンプルプログラム

まず，11行目のif文です．「if(a == b)」は，if文の始まりを表します．ここでは，変数aとbの内容を比較して，その結果をifで判断しています．多くの言語では，ifにはかならず「then」が付きものですが，C言語のifには付いていません．ifの結果が「真」(条件の計算結果が1)だった場合には，if(…)のすぐ次に書かれた実行単位が実行されます．ここでは，

```
{
printf("¥n  First == Second  ¥n");
}
```

という部分が実行単位になります.「{」と「}」の間にはいくつでもprintfなどの実行文を入れることが可能です. この例のように実行させる文が1つだけの場合は,「{」と「}」は省略して, 以下のように書くこともできます.

```
if(c == a)
    printf("¥n  First == Second  ¥n");
```

しかしC言語を使いはじめのときは, 実行単位を明確にするために「{ }」は省略せずに付けておいた方がよいでしょう. また, if文で実行される文は, 字下げ(インデント)をしておきます.

さて, ifの結果が「偽」(条件の計算結果が0)の場合は, ifの次の「else」に続く実行単位が実行されます. 形式は見てわかるとおり「真」の場合と同じです.

以上のことを整理して, 書式とフローチャートにまとめます(**図 3-3**).

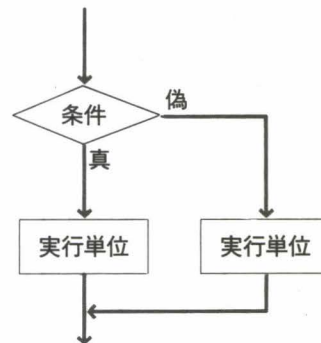



図 3-3　if文の書式とフローチャート

最後に，重ねて注意しなければならないのは，なんといっても「a == b」と，「a = b」の違いです．私もC言語を使いはじめのときはよく間違えたので，「ifが出てきたら，==を使う」というのをおまじないにしていた頃がありました．

■switch〜case文

switch〜case文は，if文とはまた違った分岐の制御を行います．if文は処理の流れを2つ(真と偽)に分けますが，switch〜case文は判断される式の値によって，2つ以上の処理にその流れを分岐させます．C言語でプログラムを書く場合，分岐の制御にはこのswitch〜case文を使うことが多く，これに習熟すると，わかりやすいプログラムが書けるようになります．

switch〜case文を使った分岐のプログラムを見てみましょう(**リスト3-10**)．

```
 1: main()
 2:      {
 3:      int    a, b, number;
 4:
 5:      printf("input <a> : ");
 6:      scanf("%d", &a);
 7:      printf("input <b> : ");
 8:      scanf("%d", &b);
 9:
10:      printf("¥n 1) a + b ¥n");
11:      printf(" 2) a - b ¥n");
12:      printf(" 3) a * b ¥n");
13:
14:      printf("¥nSelect [1, 2, 3] : ");
15:      scanf("%d", &number);
16:
17:      switch(number)
18:           {
19:           case      1:
20:                     printf("a + b = %d¥n", a + b);
21:           case      2:
22:                     printf("a - b = %d¥n", a - b);
23:           case      3:
24:                     printf("a * b = %d¥n", a * b);
25:           default:
26:                     printf("Select [1, 2, 3], retry!¥n");
27:           }
28:      }
```


[実行結果]

```
A>test↵
input <a> : 50↵
input <b> : 20↵
```

```
 1) a + b
 2) a - b
 3) a * b

Select [1, 2, 3] : 1↵  ………1番目の計算式を選ぶ
a + b = 70
a - b = 30
a * b = 1000                1番目以降のすべてのcaseが実行されてしまう
Select [1, 2, 3], retry!

A>test↵ ………もう一度やってみる
 input <a> : 50↵
 input <b> : 20↵

  1) a + b
  2) a - b
  3) a * b

Select [1, 2, 3] : 3↵  ………3番目の計算式を選ぶ
a * b = 1000
Select [1, 2, 3], retry!    3番目以降のすべてのcaseが実行されてしまう

A>
```

MSX-C, LSI C は，1行目の前に「#include <stdio.h>」を追加

リスト 3-10 switch～case 文のサンプルプログラム

見てわかるとおり，if 文とはかなり書式が違っています．その理由は，判断をする道筋が if 文の場合と比べて異なっているためです．以下にそれをくわしく説明します．

17 行目の switch 文の「( )」のなかは，if 文と同じように条件の判断を行います．つまり，〝number の値によって分岐先を決めます〃という宣言をしているわけです．次の行の「{」は，switch 文の実行単位の始まりを表し，あとに出てくる対応した「}」でくくられて，switch 文の範囲を決めています．

そして 19 行目以降の「case」で，実際に number の値が判断されます．最初の「case」では，number が〝1〃という数値であるかどうかを見ています．もし，〝1〃だったら「case　1:」のあとに続く実行文を実行します．入力された値が〝1〃でない場合には，同様に〝2〃であるか〝3〃であるかの判断が次に続く case 文でなされます．最後の「default:」は，〝どの case にも当てはまらなかった場合の case〃ということです．

さて，実行結果を見てみましょう．ある case に一致すると，それ以降の case がすべて実行されてしまい，予想した結果と違う出力が出てしまいました．これではあまり意味がありません．そこで，1つの case に一致したらプログラムを終わるように，もう1つプログラムを作ってみます(**リスト 3-11**)．

```
 1: main()
 2:      {
 3:      char     c[20]; ………文字型の配列として20文字分を確保しておく
 4:
 5:      printf(" Which is your blood-type : ");  } 文字配列c[ ]に血液型を入力
 6:      scanf("%s", c);                          }
 7:      printf("¥n"); ………1行送る
 8:                        ┌──→ if文は1行で書くこともできる
 9:      if(c[0] > 0x60) c[0] -= 0x20; ………もしc[0](1番目の文字)が小文字ならば大文字に直す
10:                                      (0x20を引いているのは,大文字のコードと小文字の
11:      switch(c[0]) ………c[0]の値により分岐     コードが0x20だけ離れているため,P.209参照)
12:           {
13:           case     'A': ………Aの場合
14:                    printf(" You are normal Japanese.¥n");
15:                    break; ………switch文から抜ける
16:           case     'B': ………Bの場合
17:                    printf(" Be careful at go-home.¥n");
18:                    break; ………switch文から抜ける
19:           case     'O': ………Oの場合
20:                    printf(" Be careful at steps.¥n");
21:                    break; ………switch文から抜ける
22:           default: ………A,B,O以外の場合
23:                    printf(" Be careful at all of your life.¥n");
24:           }
25:      }
```

[実行結果]

```
A>test⏎
 Which is your blood-type : A⏎ ………Aと入力する

 You are normal Japanese. ………break文があるので,switch文を抜け実行が終了する

A>test⏎
 Which is your blood-type : o⏎ ………小文字で入力した場合も大文字に変換されるので,
                                     caseに一致する
 Be careful at steps.
A>
```

MSX-C, LSI Cは, 1行目の前に「#include <stdio.h>」を追加

リスト 3-11 break文を伴ったswitch〜case文

今度は正しい結果が出ていますね. 13行目のcaseに続く実行文の最後に, 「break;」という文が付いています. この文が, 〝ここでswitch〜case文を終了せよ〞という意味を持っています. 正確にいうと, この文は〝現在の実行単位を抜けて, 1つ外側の実行単位に移る〞という動作を行います. **リスト3-11**ならば, 「switch(…) {……}」を抜けて「main( ) {……}」に実行が移ります. つまり, そこには何も実行文がないのでプログラムが終了することになります. 前述の**リスト3-10**のプログラムもbreak文を入れて, 正しい結果になるように変更してください.

switch 文にはかならず break 文が付きますので，いっしょに覚えておくとよいでしょう．また break 文は，他の制御文でも使用されます．

このプログラムの 9 行目については，解説を加える必要があります．この行では，入力された文字を小文字から大文字へ変換しています．209 ページのキャラクタコード表を見てください．ここで〝0x60″より大きいコードは，小文字に当たることがわかります．つまり，

```
if( c[0] > 0x60)
```

という条件は，1 文字目が小文字かどうかを判断することです．そして，

```
c[0] -= 0x20;
```

という実行文は，小文字と大文字のコードの差(0x20)をとって大文字に変換しています．

ちょっと横道にそれてしまいましたが，**図 3-4** に switch～case 文の書式とフローチャートをまとめておきます．

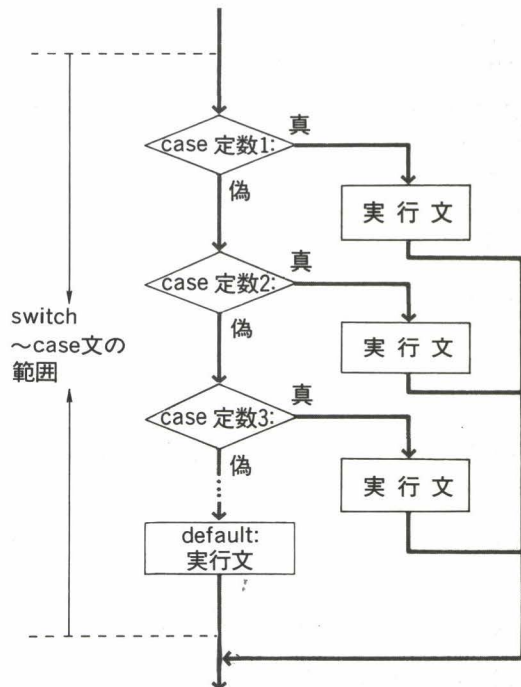



図 3-4 switch～case文の書式とフローチャート

■while文

whileという単語は，「～をしている間」という意味があります．ここで示すwhile文と次で取り上げるfor文は，C言語でループを形成する制御文です．この2つを知っていれば，BASICなどのgoto文を使うよりも，ずっとわかりやすくループの制御を行うことができます．まず，while文から見てみましょう(**リスト3-12**)．

```
 1: main()
 2:      {
 3:      int   i, j;
 4:      i = 0;
 5:      printf("Number : ");
 6:      scanf("%d", &j);
 7:                   ──→ 条件
 8:      while( i < j )………i<jが真の間,実行単位を繰り返す(セミコロンは付けない)
 9:           {
10:           printf("** %d **¥n", i);       while文の実行単位
11:           i++; ………iの値を1増やす
12:           }
13:      }
```

[実行結果]

```
A>test↵
Number : 5↵
** 0 **
** 1 **
** 2 **
** 3 **
** 4 **
A>
```

MSX-C，LSI Cは，1行目の前に「#include　<stdio.h>」を追加

リスト3-12　while文のサンプルプログラム

リスト3-12の「while」のある行から説明します．「while(i ＜ j)」は，〝変数iの値が変数jの値よりも小さいうちは，以下の実行文を繰り返せ〟という意味です．その実行文とは，もちろんこの行の後に続く「{」と「}」に囲まれた部分のことです．そのなかでは，printfで変数iの内容を表示させた後，「i＋＋;」としてiに1を加えています．つまり，これらは全体で〝iを0からjを越えない数まで数えて，それを表示しろ〟ということになります．

前の節のif文やswitch文でも同じですが，「( )」のなかの論理式の値は，「真」の場合は〝1〟であり，「偽」の場合は〝0〟ですから，**リスト3-13**のように「while(1)」とすると，無限にループを繰り返します．

```
1: main()
2:     {
3:     while(1) ………条件は常に真なので，無限ループを形成する
4:         {
5:         printf("I love C¥n");
6:         }
7:     }
```

［実行結果］

```
A>test↵
I love C
I love C
I love C
I love C
I love C
    ⋮
CTRL + C で実行を中断する
A>
```

MSX-C，LSI C は，1行目の前に「#include　<stdio.h>」を追加

リスト 3-13　while 文による無限ループ

これは，長さがわかっていないファイルの中身を取り出す場合とか，形式に合った入力がされるまで実行するという場合に，かなり頻繁に使われる方法の1つです．

while 文の書式とフローチャートを**図 3-5** に示します．

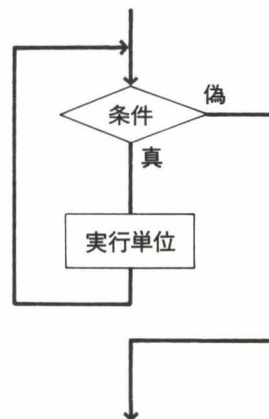



図 3-5　while 文の書式とフローチャート

■ for 文

さて，何といっても制御文のなかでの使用率ナンバーワンは，この for 文です．for 文は，while 文よりも機能が多く複雑なループの制御を1文で行うことができます．

サンプルプログラムを見てみましょう．ここでは**リスト 3-12** の while 文と同じことをやっています(**リスト 3-14**)．

```
 1: main()
 2:         {
 3:         int     i, j;
 4:
 5:         printf("Number : ");
 6:         scanf("%d", &j);
 7:
 8:         for(i = 0 ; i < j ; ++i)
 9:                 {
10:                 printf("** %d **¥n", i);
11:                 }
12:         }
```

セミコロンは付けない

while文と比べて，1文で記述できる

for文の実行単位

[実行結果]

```
A>test ↵
Number  : 5 ↵
** 0 **
** 1 **
** 2 **
** 3 **
** 4 **
A>
```

MSX-C，LSI C は，1行目の前に「#include　<stdio.h>」を追加

リスト 3-14　for 文のサンプルプログラム

for の後の続く「( )」のなかに，いくつもの式が並べて書いてあります．これによって**リスト 3-12** に比べると，非常にすっきりとまとまっています．

図 3-6 にそのなかの構成要素を示します．要素は3つあって，それぞれは「;」で区切られています．ループを構成するために必要な要素である〝初期値の設定〟，〝条件〟，〝ループのカウンタ〟を一度に記述することができるので非常に便利です．ループのほとんどは，この for 文で記述することが可能です．

また，この中身はそれぞれ省略することもできます(ただし，セミコロンは省略できません)．

```
for( i = 0 ; i < 10 ; ++i )
```

- ++i ― ループを一回まわるごとにこの式を実行する
- i < 10 ― この式の値が「真」の間，実行単位を繰り返す
- i = 0 ― ループに入る前にこの式を実行する

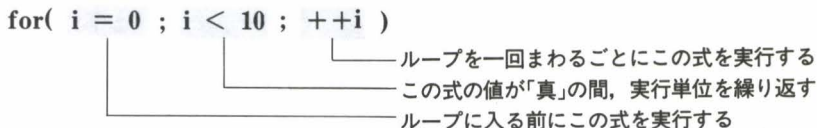



図 3-6 for 文の制御変数

8 行目 for 文は，〝i を 0 から j まで，1 つずつ増やしながら以下の実行文を繰り返せ〟ということになります．

では**リスト 3-15** のような for 文は，いったい何を意味するのでしょうか？

```
1: main()
2:     {
3:     for( ; ; )
4:         {
5:         printf("I love C¥n");
6:         }
7:     }
```

（注：セミコロンは省略できない）
（注：forの( )のなかが空文になり，無限ループを形成する）

[実行結果]

```
A>test
I love C
I love C
I love C
I love C
I love C
   ⋮
```

CTRL + C で実行を中断する

```
A>
```

MSX-C，LSI C は，1 行目の前に「#include <stdio.h>」を追加

リスト 3-15 for 文による無限ループ

決してミスプリントではありません．これは〝とにかくまわれ〟，すなわち無限ループを作ります．また，このプログラムの「for(…)」の部分はそのままにして(つまり for 文での無限ループを使って)，**リスト 3-14** と同じことをさせてみます．ここでは，前に使った if 文と break 文が登場します．

```
 1: main()
 2:      {
 3:      int    i, j;
 4:
 5:      printf("Number : ");
 6:      scanf("%d", &j);
 7:      i = 0;
 8:      for( ; ; ) ………無限ループを作る
 9:           {
10:           if(i < j) ………i<jが真かどうかの判断
11:                {
12:                printf("** %d **¥n", i); ………変数iの表示
13:                i++; ………変数iに1を加える
14:                }
15:           else ………i<jが偽である場合
16:                {
17:                break; ………for文の実行単位を抜ける
18:                }
19:           }
20:      }
```

[実行結果]

```
A>test↵
Number : 5↵
** 0 **
** 1 **
** 2 **
** 3 **
** 4 **
A>
```

MSX-C, LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト 3-16　リスト3-14を別の形で実現する

リスト 3-16のように書くこともできますが，このような書き方をおすすめできないのはいうまでもありません．

最後に，for文の書式とフローチャートをまとめておきます(**図 3-7**)．

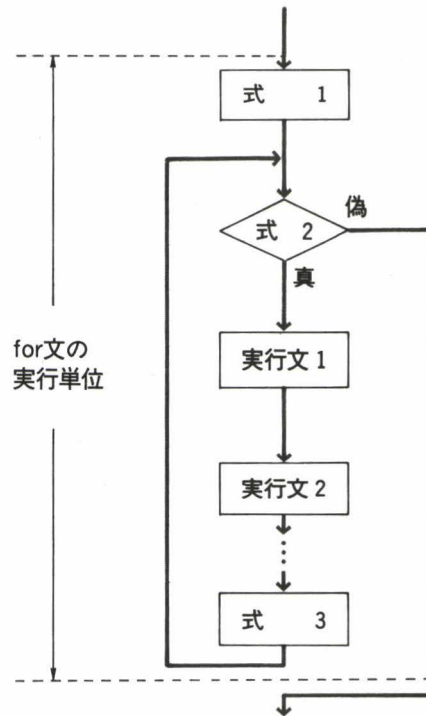



図 3-7　for 文の書式とフローチャート

■ その他の制御文

さて，このほかにもC言語は do～while 文，goto 文という制御構造があり，また補助的な制御文として continue 文，return 文を持っています．

これらの紹介していないもののうち，「return 文」と「continue 文」については，4章と5章にでてきます．

また，「do～while 文」は，個人的な趣味の問題や他言語での慣れの問題として，かなり多用している人もいるようです．これからC言語を始める場合は，本章で紹介した制御構造だけですべてのプログラムの記述が可能なので，それらを使ってプログラムを組むことをおすすめします．

それから「goto 文」は〝C言語の癌〞とも呼ばれています．まず絶対に使わないでください．ここで説明した制御構造を用い，プログラムの設計がしっかりしていれば goto 文は必要ありません．

■制御構造のまとめ

まとめとして，これまで示した制御文を組み合わせたプログラムを示しておきます．ここで紹介するのは，「数当てゲーム」です．プログラムについての解説は省略しますが，for 文，if 文，switch～case 文が使われていますので，その書式や処理の流れを確認してください．

このプログラムのフローチャートを**図 3-8** に，プログラムリストを**リスト 3-17** に示します．

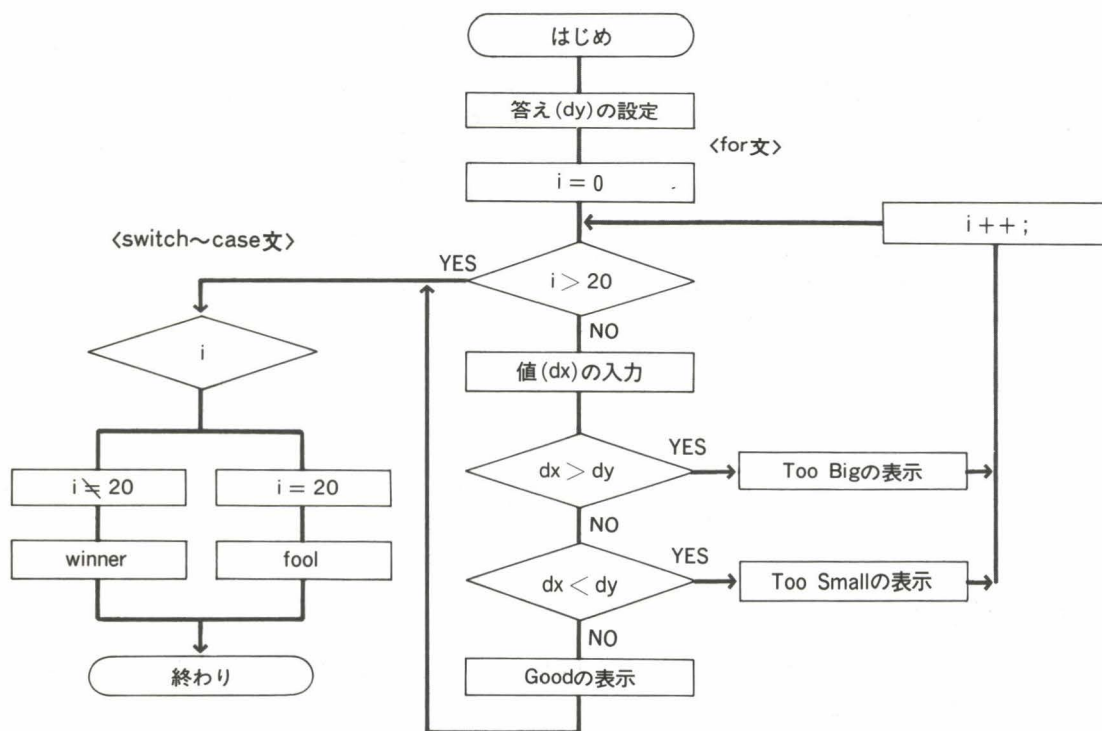



図 3-8　数当てゲームのフローチャート

```
 1: main()
 2:     {
 3:     int   dx, dy, i;
 4:
 5:     dy = 3453;………これが答え,乱数を使って決めるとよい(5章の標準関数参照)
 6:
 7:     for(i = 0 ; i < 20 ; i++) ………チャンスは20回与えられる
 8:         {              ┌──現在が何回目のトライか
 9:         printf("¥n(%d) Input Number : ", i + 1);  } 予想した数の入力
10:         scanf("%d", &dx);                         }
11:
12:         if(dx > dy)                               } 入力した数が大きかった場合
13:             printf("¥n Too big! retry.¥n");       }
14:         else
15:             if(dx < dy)                           } 入力した数が小さかった場合
16:                 printf("¥n Too small! retry.¥n"); }
17:             else
18:                 {                                 } 上記以外の場合
19:                 printf("¥n Good! right number.¥n");} (すなわち正解)
20:                 break;………正解の場合はfor文のループを
21:                 }                 抜ける
22:         }
23:
24:     switch(i) ………20回のトライのうちに当たったかどうかの判断(ここでは2つの条件判断に switch文を使っている)
25:         {
26:         case    20: ………20回のトライで当たらなかった場合
27:                 printf("¥n ** You are fool good-by! **¥n");
28:                 break;
29:         default:………20回以内で当たった場合
30:                 printf("¥n ** You are winner by %d try **¥n", i + 1);
31:         }                                                 ↑
32:     }                                                 何回目のトライで
                                                          当たったか
```

[実行結果]

```
 A>test↵ ………プログラムの実行

 (1) Input Number : 2000↵ ………1回目は小さすぎたようだ
 Too small! retry.

 (2) Input Number : 4000↵ ………2回目はちょっと大きすぎた
 Too big! retry.

 (3) Input Number : 3000↵
 Too small! retry.
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   Input Number : 3452↵
   small! retry.
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(12) Input Number : 3453↵ ……12回目でやっと当たり!
 Good! right number.

 ** You are winner by 12 try **
 A>
```

MSX-C, LSI C は, 1行目の前に「#include <stdio.h>」を追加

リスト 3-17 数当てゲームのプログラムリストと実行結果

【この章のポイント】

1．scanf の使い方
2．演算子はほとんど今までのものと同じ．ただし，以下のものに注意

==	［等しい］	!=	［等しくない］		
&&	［論理積(AND)］	¦¦	［論理和(OR)］	!	［否定(NOT)］

3．if，for，while，switch 文の形式を覚える
4．switch 文には，かならず break 文が入る

［練習問題］

1．次の if 文で書かれたプログラムを switch 文を使って書き直しなさい

```
 4:            :
 5:     int i, a;
 6:     for(i = 0 ; i < 20 ; i++)
 7:         {
 8:         if(a == 1)
 9:             {
10:             printf("*** Devil Appeared !¥n");
11:             break;
12:             }
13:         if(a == 2)
14:             {
15:             printf("*** There are many angels !¥n");
16:             break;
17:             }
18:         if(a == 4)
19:             {
20:             printf("*** The Dragon's sun is dead.¥n");
21:             break;
22:             }
23:         else
24:             printf("*** You are dead Good-bye !¥n¥n");
25:         }
```

2．次のプログラムは number で指定した文字列を 1 文字ずつ表示するプログラムだが正しい結果が出力されない．このプログラムを for 文で書き換えなさい

```
 1: static char number[21] = "12345678901234567890";
 2:
 3: main()
 4:     {
 5:     int     i;
 6:     i = 0;
 7:
 8:     while(20 > ++i)
 9:         {
10:         printf("Number is %c¥n", number[i]);
11:         }
12:     }
```

MSX-C，LSI C は，1 行目の前に「#include <stdio.h>」を追加
BDS C は，「1： char *number；　number = "1234567890123456789"；」に変更

%Cは1文字を表示する表現指示のための文字

（解答は217ページ）

第4章

よりC言語らしいプログラミング

2章，3章では，C言語プログラミングの基礎を学んできました．この章では，C言語の特徴をより理解してもらうために，「C言語でしかできないプログラミング」を中心に解説します．

C言語は前にも述べたように，〝さまざまなコンピュータの機種の違いにかかわらず同じプログラムが書ける〟，〝コンピュータのきめ細かい制御が可能である〟という，2つの相矛盾する問題を解決したことによって，この世の中で認められるようになった言語です．

きめの細かい制御をしようと思えば，個々のコンピュータに合ったプログラムを書かなければなりません．また，どの機種でも同じプログラムが動くようにするには，特定の機種でしかできないものを扱うわけにはいきません（つまり，きめの細かいプログラミングができません）．

このことは人間でいえば，同じ内容の事柄を相手に伝えるのに，アメリカ人には英語で，日本人には日本語で話さなくては話の内容を理解してくれないのと同じことです．どんな人にもまったく同じ言葉で，同じ内容が伝わるわけではありません．ところが，アメリカ人も日本人も同じ人間であることには変わりませんから，どんな人とも同じ言葉で話せた方がよい場合もあります．コンピュータの世界でもこの事情は同じです．

人間の世の中では，たとえば〝エスペラント〟のようなどの国の文化ともかかわりの薄い言葉を設定することによって，この問題を解決しようと試みた人たちがいます．コンピュータの世界ではC言語が，人間社会のエスペラントのような役割を担いつつあるのではないかと思います．

4.1 コンピュータの基本的な構造

C言語は，どこでも，いつでも同じように扱える言語です．この言語を理解するには，現在あるコンピュータ(大型コンピュータから，MSXのような小さなものまで)がいったいどんな機構でどのように動いているのか，またそれらはどこが似かよっていてどこが違うのかということを考える必要があります．つまり，どの機種でも同じものとは何かを知らなければなりません．とはいっても，全部を理解する必要はなく，ごく本質的なことをほんの少し覚えていればよいのです．

ここでは，C言語をより深く知ってもらうために〝コンピュータの基本的な構造〟について説明していきます．

■コンピュータに共通する要素

コンピュータといっても，小はゲーム専用のものから，大は億単位という値段の大型機まであり，その種類も私たちが使うことの多いパソコンから，多くの人々の目には触れることのない駅の自動販売機のなかの制御用コンピュータまで，実にさまざまなものがあります．C言語は，このようなコンピュータのいずれのプログラムも書けるように考えられ，時代と共に練られてきました．

さて，これらのコンピュータすべてに共通した要素はいったいなんでしょうか．それは，**メモリ**と**CPU**(Central Processing Unit：中央処理装置)です．この2つは，現代のコンピュータの核を形成するものといえます．

まずはこの2つのかかわりから見て行くことにしましょう．

■データとプログラムの親密な関係

メモリとは，〝1〟と〝0〟の電気信号になった情報を蓄えておくところです．「プログラム記憶方式(Stored Program Architecture)」と呼ばれている現在のコンピュータは，プログラムもデータもすべて同じメモリ上に蓄えます．

CPUがプログラムを実行するには，メモリに記憶させてあるプログラムを読みにいき，その指示にしたがって〝どこのメモリにあるデータをどう処理するか〟を決め，その指示どおりにデータの処理を行います．

以下の**図4-1**にメモリとCPUの概念図を示します．

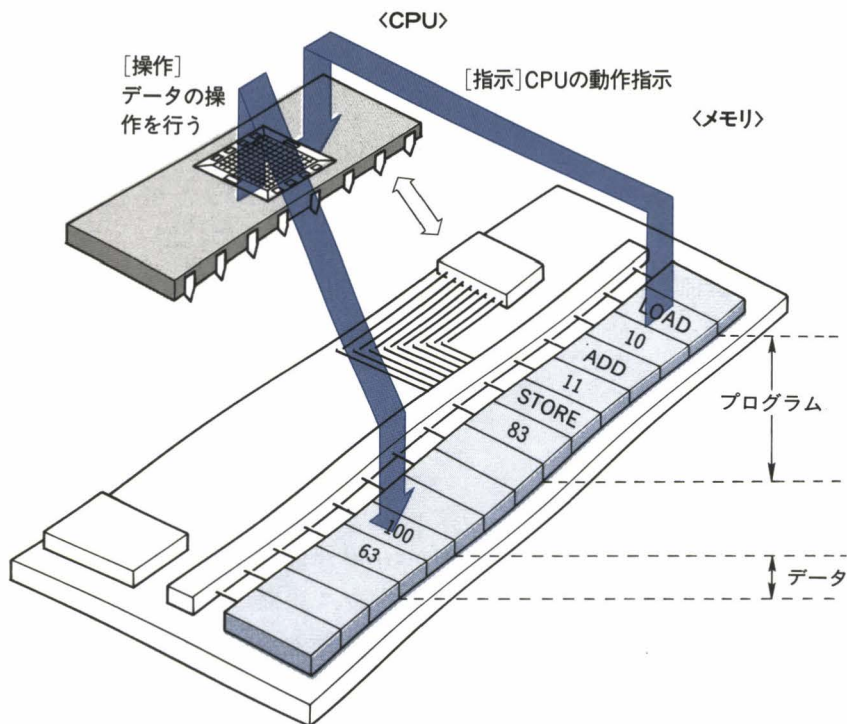



図 4-1　メモリとCPU

■コンピュータはいかにして暴走するか

つまり，メモリ上のデータとプログラムはまったく同じ形をしており，〝これはデータ〟，〝これはプログラム〟という分け隔てはいっさいありません．CPUは，メモリ上の特定の位置から始まる一群のデータを勝手に〝これはプログラムだな〟と解釈し，実行を始めるようになっています．つまり，メモリ上の位置によって，データとプログラムを分けているにすぎません．

もし，プログラムのあるべき場所にデータが入っていると，CPUはそのデータをプログラムと勘違いして，めちゃくちゃなプログラムを実行してしまいます．これを**暴走**と呼びます．

■コンピュータのなかの「住所」

データとプログラムを区別するために，〝メモリなかの場所を指定する〞ということは，コンピュータを動かす上で最も重要な作業です．そして，どこのメモリを指定するか，どこのメモリからプログラムを実行するか，どこのメモリには何を意味するデータが入っているかなどの情報を管理できるように，メモリの1つ1つには番号が振ってあります．この番号のことを**アドレス**(Address)と呼びます(日本語に直せばまさに〝住所〞ですね)．

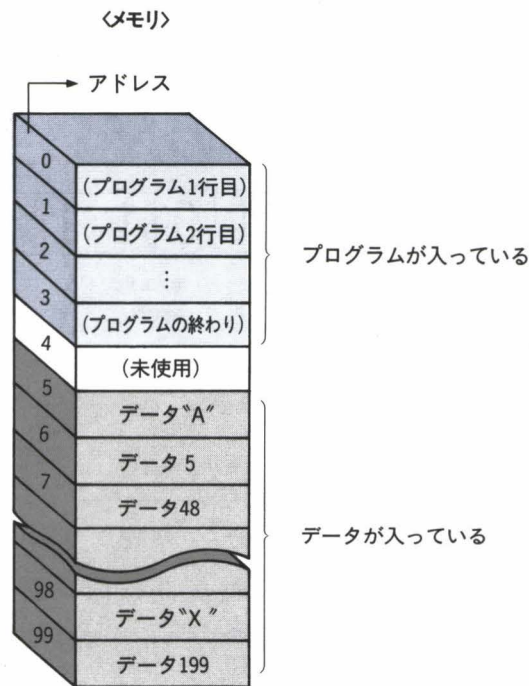



図 4-2　アドレス

図 4-2 では，例としてアドレス0から3にプログラム，アドレス5から99にデータが入っているものとしています．実際のコンピュータでは，何千，何万といった単位の数のメモリがあります．

■コンピュータの世界は常に８ビット

次に，もっとくわしくアドレスを持つメモリの１つに，どのように情報が記憶されているのかを見てみましょう(**図 4-3**).

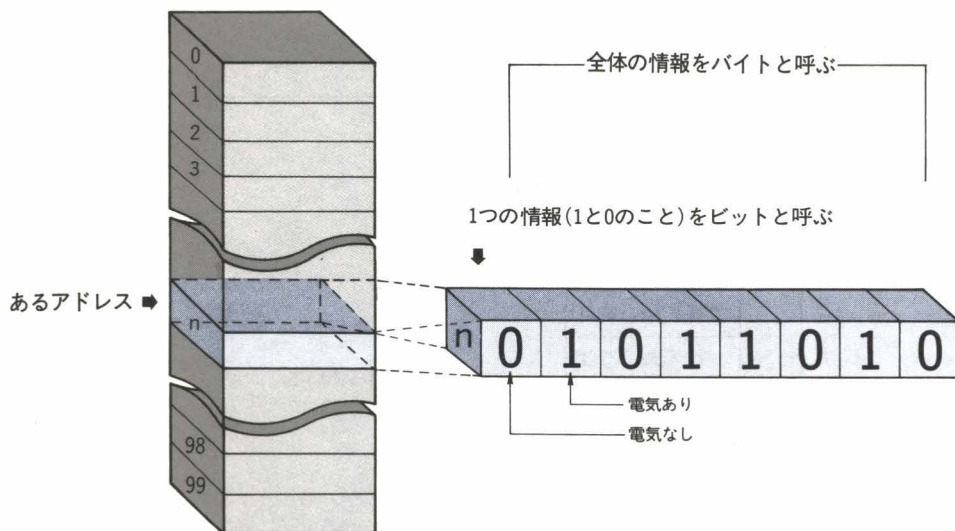



図 4-3　1つのメモリの中身

図 4-3 のように電気の"ある/なし"の２つの状態を保つことができる素子が，８個並んでいます．この素子の１つを「メモリ・セル(Memory-Cell)」と呼びます．日本語に直せば「記憶細胞」といったところです．

このセルには，電気のある状態を"１"，電気のない状態を"０"として情報が蓄えられています．これを**ビット**(bit)と呼びます．さらに，この１と０の８個の並びのことを，**バイト**(byte)と呼びます．つまり，１バイトの情報(１と０の８個の集まり)は１つのアドレスを持っていることになります．

人間の世の中では，１つの住所(アドレス)に住んでいる１世帯の家族構成(ビット数)はまちまちですが，コンピュータの世界の１世帯の家族構成は常に同じ(８ビット)なのです．

■変数と変数名が使えると

C言語では，これらのアドレスに人間がわかりやすいように「名前」を付けて処理を行うことが可能です．たとえば，〝アドレス98番はcandy〟，〝アドレス7番はpig1〟などとしておきます．この名前のことを変数名，変数名で示されるメモリのことを変数といいます(**図4-4**)．

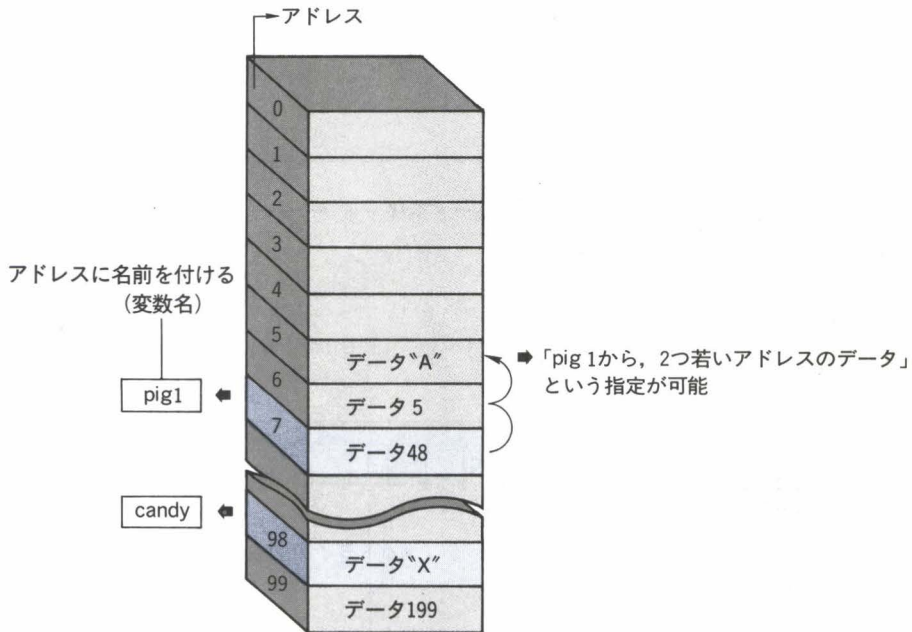



図 4-4　アドレスと変数名

つまり，〝南青山5-11-5に住む人〟というわかりにくい呼び方(アドレス)ではなく，〝東さん〟というわかりやすい名前で呼べるわけです．また，〝pig 1から，2つ若いアドレスのデータ〟という指定もできます．前出の表現を借りれば，〝東さんの家の2軒手前の人〟といった表現になります．具体的には4.3節で実際にやってみます．

■周辺機器のコントロール

さて，私たちが通常触っているコンピュータは，パソコンも大型も含めて，キーボードやディスプレイやフロッピーディスクなどの〝手足〟を持っています．CPUは，このような周辺機器もすべてメモリと同じ扱いをしています．たとえば，**図4-5**のようにフロッピーディスクの入出力や制御のための情報は，普通のメモリと同じように配置されているにすぎません．そして，これらの特別なアドレスで示されるメモリのビットは，〝1〟を書くとフロッピーディスクのモーターが回り出すというような「スイッチ」の役目を果しているのです(実際はもっと複雑なのですが，一応そう理解しておいてください)．

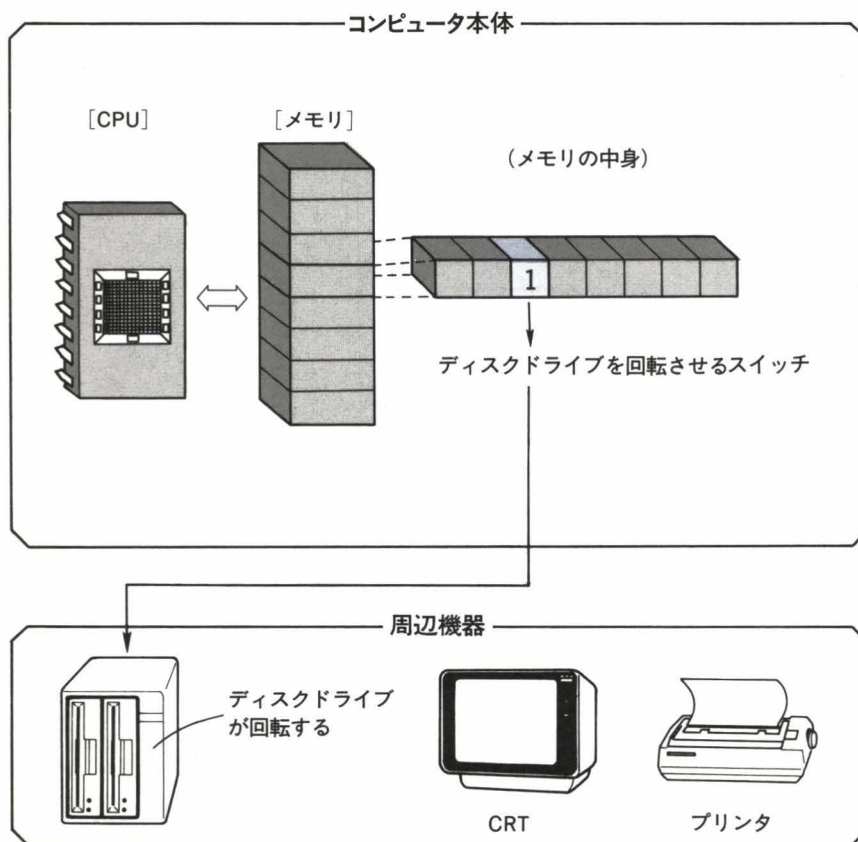



図 4-5　周辺機器の動作の制御

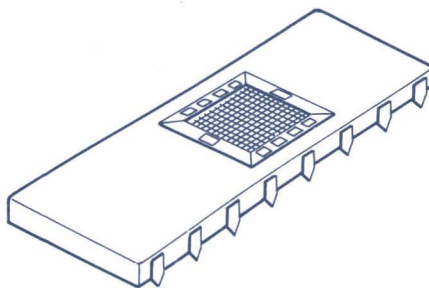
■メモリの操作とC言語

これらのメモリ操作がとくに重要となるプログラムの代表的なものに，Video-RAM(画像メモリ)の直接アクセスがあります．この技術は，高速性を要求されるグラフィックスやワードプロセッサなどのアプリケーションに利用されています．

C言語では，メモリのアクセスを「ポインタ」という概念を使って行うことができます(ポインタについては4.3節で取り上げます)．メモリの操作などは従来はアセンブラでしか書けなかったものですが，C言語を使えば高級言語で記述が可能になります．

16ビットCPU or 8ビットCPU!

1つのアドレスを持つメモリの構成が8ビット(1バイト)というのは，代表的なパーソナル・コンピュータの場合を示しています．大型のコンピュータでは，この8ビットという数字が36ビットになったりします．最近の16ビットCPUというのは，メモリ上のデータを処理する単位が16ビットであるということです．世の中には，ひねくれた技術者(失礼，ほんとうはそういったことが必要だからなのですが)もいて，この数字を自由に替えられるコンピュータも存在します．

また，PC-9801などに使われている8086CPUやV30CPUなどは，1つのアドレスが持っているビット数は8なのに，隣同士の2つ分のアドレスのメモリの中身をいっぺんに処理できてしまうため，〝16ビットCPU〟と称しています．ということは，本書での「～ビットCPU」の定義は「1つのアドレスに何ビット使っているか」によるので，8086CPUは，本書では8ビットCPUということになってしまいます?!

4.2　C言語のメモリの使い方 ―記憶クラス―

前の節で，一般的なコンピュータの構造(特にメモリの使い方)を学んできました．C言語は，メモリを自在に使ったプログラムが組めるように設計されていることが特徴です．この点は，BASICなどの他のプログラミング言語と大きく異なるところです．

この節では，C言語におけるデータの扱いとそれが実際のメモリ上でどう扱われるかを中心に見ていきます．

■C言語における変数の種類

C言語の変数には，2章で解説した型(int，long，charなど)の概念とは別に，もう1つ重要な概念があります．この概念は，C言語のプログラム単位である**関数**と密接なかかわりを持っています．関数については，以降の4.4節で解説しますが，ここではサブルーチンのようなものだと思っておいてください．

C言語で使う変数の種類を**表4-1**に示します．

名称	意味
auto	共有エリアに，一時的にメモリを割り当てられる変数
static	一度メモリを割り当てられると，プログラムの最初から最後まで同じアドレスに存在する変数
†extern	他のプログラム単位にある変数
†register	CPUのなかのメモリ(レジスタ)を割り当てる変数

注)BDS Cは，static変数を使えません

表4-1　C言語の変数の種類

データの型は，変数で扱うことのできる数値の大きさを決めていました．**表4-1**の種類では，メモリ中のどこに変数の領域が割り当てられるかを決定します．これは，たとえばループ中のカウンタのようにあるところだけで使う変数と常に記憶しておく変数を別の領域に置くということです．この区別をすることでメモリの有効活用が図れます．

† extern変数とregister変数は，本書では取り上げません

また，これらの変数は実行単位（{ }で囲まれた範囲）という観点から，さらに以下の２つに分けることができます．

ローカル変数　……　宣言された実行単位のなかでだけ使用可能

グローバル変数　…　すべての実行単位で使用可能

この区別は，変数を宣言した位置によって決まります（**図 4-6**）．つまり，ある実行単位のなかで宣言すれば，そのなかでのみ有効な変数となり，実行単位の外で宣言するとどこでも参照できる変数となります．BASICの変数はどこでも参照できるのでグローバル変数ですが，C言語ではローカル変数があることによって，各実行単位ごとに変数の独立が保たれています．

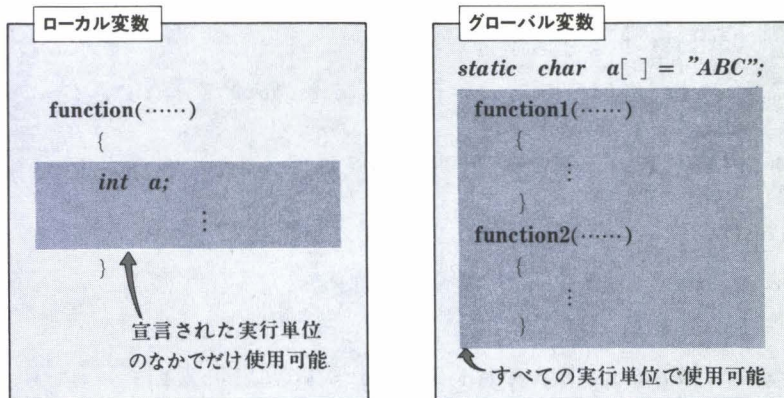



図 4-6　ローカル変数とグローバル変数

以下に，それぞれの変数について解説していきます．これまで述べたように，C言語では実行単位というまとまりが非常に重要な意味を持っていますので，この点をとくに注目してください．

■ auto 変数

まずは，auto変数から見てみましょう．この「auto」という文字は普通のC言語のソースファイル中では見られません．「auto」とは〝自動〟ということですから，とにかくただ「char　a;」のように宣言すると，これは暗黙のうちにauto変数ということになります．

auto変数は，メモリの共有エリアに一時的にそのアドレスがとられます．そのためローカルで宣言した場合には，一度実行単位を抜けるとその変数の領域は消え，別の変数がまたその領域を利用します．

◆ローカルで宣言した場合

C言語のプログラムのなかで最も多いのが，auto変数でかつローカル変数の場合です．以下に，サンプルプログラムを示します(**リスト 4-1**)．

```
                                          変数 a, b, i の有効範囲
 1: main()
 2:     {
 3:     int a, b, i; ………関数main( )で宣言している変数a, b, i
 4:
 5:     a = 2; b = 3;
 6:
 7:     printf("¥n main-a = %d¥n main-b = %d¥n", a, b);
 8:
 9:     for(i = 0 ; i < a ; ++i)              変数jの有効範囲
10:         {
11:         int j; ………1番目のfor文のなかで宣言している変数 j
12:
13:         for(j = 0 ; j < b ; ++j)
14:             {
15:             printf(" main-Loopl + main-Loop2 = %d¥n", i + j);
16:             }
17:         }
18:     function(); ………関数functionの実行
19:     }
20:
21: function()                               変数a, bの有効範囲
22:     {
23:     int a, b; ………関数function( )で宣言している変数a, b
24:                    (関数mainの変数a, bとは別のもの)
25:     a = 1; b = 2;
26:
27:     printf("¥n func-a = %d¥n func-b = %d¥n", a, b);
28:     }
```

[実行結果]

```
A>test⏎

 main-a = 2  } 関数mainの変数a,b
 main-b = 3  }
 main-Loopl + main-Loop2 = 0  ………i=0, j=0
 main-Loopl + main-Loop2 = 1  ………i=0, j=1
 main-Loopl + main-Loop2 = 2  ………i=0, j=2
 main-Loopl + main-Loop2 = 1  ………i=1, j=0
 main-Loopl + main-Loop2 = 2  ………i=1, j=1
 main-Loopl + main-Loop2 = 3  ………i=1, j=2

 func-a = 1  } 関数functionの変数a,b
 func-b = 2  }
A>
```

MSX-C，LSI Cは，1行目の前に「#include　<stdio.h>」を追加
RUN/Cは，ローカル変数を関数の最初で宣言する必要があるので，11行目を削除し「3:　int　a, b, i, j;」に変更

リスト 4-1　auto変数で，かつローカル変数の場合

リスト 4-1 では，関数ごとに変数 a と b が同じ名前で宣言されているのに，違う変数であることを示しています．また，変数を宣言した位置によって，その有効範囲が異なっています．これは，各変数がそれぞれの実行単位のなかでだけ使われるローカル変数であるためです．この点は BASIC でいう変数とは大きく異なっています．

◆グローバルで宣言した場合

auto 変数でかつグローバル変数の場合は，各関数で共通にその変数を使うことができます．**リスト 4-2** にサンプルプログラムを示します．

変数a, bの有効範囲

```
 1: int a, b; ………関数の外で宣言された変数a, b
 2:
 3: main()
 4:      {
 5:      a = 2; b = 3; ………グローバル変数として宣言されているので，関数内で宣言
 6:                           する必要はない
 7:      printf("¥n main-a = %d¥n main-b = %d¥n", a, b);
 8:
 9:      function(); ………関数functionの実行
10:      }
11:
12: function()
13:      {
14:      a = 1; b = 2; ………関数mainと同じ変数を使っている
15:
16:      printf("¥n func-a = %d¥n func-b = %d¥n", a, b);
17:      }
```

[実行結果]

```
A>test ⏎

 main-a = 2
 main-b = 3
                } 関数mainと関数functionで同じ変数に異った値を代入して表示させている
 func-a = 1
 func-b = 2
A>
```

MSX-C，LSI C は，1 行目の前に「#include　<stdio.h>」を追加

リスト 4-2　auto 変数で，かつグローバル変数の場合

この例では，変数 a と b が main と function という 2 つの関数で共に使われています．static 変数で宣言した場合との違いは，初期化を行えないこととプログラムを ROM 化するときに RAM 上にその領域が確保されることです．

■ static 変数

static 変数は，auto 変数と違いそれぞれの変数に１つのメモリ領域が確保されます．このため static で宣言すると，プログラムが動く直前に変数に決まった値や文字をセットすることが可能です．このことを（値の）**初期化**といいます．

◆ローカルで宣言した場合

ローカル変数で宣言した場合は，その実行単位でのみ利用することができます．static 変数は各々に１つのメモリ領域が確保されますから，実行単位を抜けた後も値が保存されます．**リスト 4-3** にサンプルプログラムを挙げます．

```
 1: main()
 2:      {
 3:      int i, j;
 4:
 5:      for(i = 0 ; i < 2 ; ++i) ➡A
 6:           {
 7:           static int sx = 2; ………static変数で宣言し,2で初期化する
 8:
 9:           for(j = 0 ; j < 2 ; ++j) ➡B
10:                {
11:                printf(" Loop1 + Loop2 = %d¥n", sx + j);
12:                sx++;
13:                }
14:           }
15:      }
```

変数sxの有効範囲

［実行結果］

```
A> test↵
 Loop1 + Loop2 = 2    ………sx = 2, j = 0 ┐1回目のAのループ
 Loop1 + Loop2 = 4    ………sx = 3, j = 1 ┘
 Loop1 + Loop2 = 4    ………sx = 4, j = 0 ┐2回目のAのループ
 Loop1 + Loop2 = 6    ………sx = 5, j = 1 ┘

A>
```

→ Bのループを抜けた後も変数sxの値が保存されている

```
 7:           int sx = 2;   ………7行目をauto変数に変更してコンパイルし直す
```

［実行結果］

```
A>test↵
 Loop1 + Loop2 = 2    ………sx = 2, j = 0 ┐1回目のAのループ
 Loop1 + Loop2 = 4    ………sx = 3, j = 1 ┘
 Loop1 + Loop2 = 2    ………sx = 2, j = 0 ┐2回目のAのループ
 Loop1 + Loop2 = 4    ………sx = 3, j = 1 ┘
A>
```

→ Bのループを抜けると変数sxの値は消えてしまう

MSX-C，LSI C は，１行目の前に「#include　<stdio．h>」を追加
BDS C は，static 変数がないので動作しない

リスト 4-3　static 変数で，かつローカル変数の場合

◆グローバルで宣言した場合

グローバル変数として宣言すると，各関数のどこでもこの変数を利用することができます．BASICの変数はすべてがこの場合に当たります．autoで宣言した場合との違いは，値の初期化を行えるという点です．以下にサンプルプログラムを示します(**リスト 4-4**)．

```
 1: static  char    a[27] = "Better half is bitter half";  } 変数の宣言と
 2: static  int     b = 30;                                } 初期化を行う
 3:
 4: main()                                          ← 変数a[ ]とbが
 5:      {                                             有効な範囲
 6:      printf("¥n From main = %s¥n", a);
 7:      printf(" From main = %d¥n", b);
 8:
 9:      sub(); ………関数subを呼び出す
10:      }
11:
12: sub()
13:      {
14:      b = b + 1;
15:
16:      printf("¥n From sub  = %s¥n", a);
17:      printf(" From sub  = %d¥n", b);
18:      }
```

[実行結果]

```
A>test⏎

 From main = Better half is bitter half  }
 From main = 30                          }
                                         } 同じ変数を関数mainとsubで表示
 From sub  = Better half is bitter half  }
 From sub  = 31                          }

A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加
BDS Cは，「1: char *a;」，「2: int b;」に変更し，5行目の後に「a = "better half is bitter half"; b = 30;」

リスト 4-4 static変数で，かつグローバル変数の場合

よく他の言語に慣れた方がC言語で初めてプログラムを書くときに，すべての変数をstaticで使う例を見かけますが，C言語の正しい使い方ではありません．static変数を使うのは，変数をあらかじめ初期化しておきたい場合や実行単位を抜けた後もその変数の値を記憶しておきたい場合です．

■メモリの使い方のまとめ

これまでに解説した変数が，メモリ上でどのように扱われるかをまとめて**図4-7**に示します．C言語では，ここで解説したメモリの使い方の分類を**記憶クラス**と呼んでいます．

よく〝C言語はできあがった実行プログラムが他の言語に比べて小さい〞と言われていますが，その原因の1つはメモリの領域を共有できるauto変数の存在にあります．

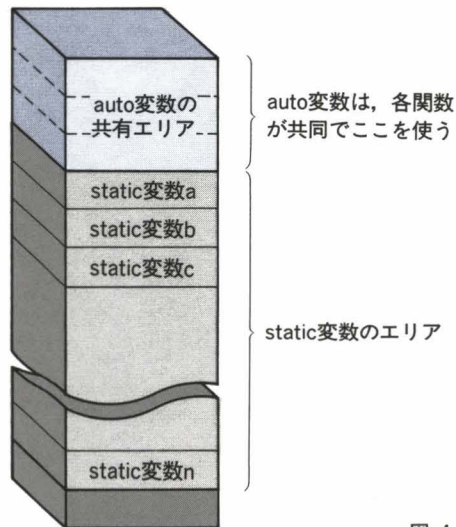



図 4-7　記憶クラスのまとめ

ある変数をautoにするのかstaticにするのか，ローカルで扱うかグローバルで扱うかという問題は，プログラムを組む上での1つのキーポイントになります．

これまでBASICやFORTRANなどの高級言語では，〝変数はなるべくその言語の使用者に意識させてはいけない〞という暗黙の約束事があったように思います．ところがこのC言語はその逆で，変数の成り立ちをプログラマは知らなくてはなりません．これが，〝C言語なんて構造化されたアセンブラじゃないか．アセンブラに毛が生えたようなものさ〞と，陰口を叩かれる理由の1つです．しかし，このようなデータの扱いができるからこそ，高級言語できめの細かい処理が可能なのです．

4.3　ポインタ

4.1節の「コンピュータの基本構造」で説明したように，コンピュータのなかのメモリはすべて「アドレス」という一連の番号によって管理されています．この番号は1つのコンピュータのなかでは決して動いたりすることはありません．たとえば〝アドレス1234番のメモリ〟は，その同じ番号を使ってメモリのなかをのぞいたり内容を書き換えたりする限り，いつも同じメモリがその対象になります．

これは非常にわかりやすいのですが，同時に不便な面もあります．メモリの位置がすべて数字で表されるので，この値を少しでも間違うとまったく違ったメモリを読み書きしてしまうことになりますし，何よりも私たちは数字の洪水のなかでプログラミングしなければなりません．

C言語では，このようなアドレスの操作の問題をポインタという概念を使って解決しています．このポインタの概念は，C言語で初めて採用されたわけではありません．コンピュータ自身が，すでにこの概念を実現するようにできあがっているのです．

では，この「ポインタの世界」を探ることから始めましょう．

■ポインタの一般的な考え方

まず，ポインタという名称を考えてみます．これは英語では〝pointer〟となるわけで，「指し示すもの」といった意味です．C言語では，これは「データを指し示すもの」と考えることができます．

データの処理を行う場合，CPUはどのデータを処理の対象にするのかをわかっていなければなりません．このとき，〝このデータ〟というのを直接指定するのに**アドレス**が使われるわけですが，C言語では，ここでポインタを使用することによってアドレスを自在に，そしてわかりやすく扱うことができます．いったいどのように，このポインタを使うのかを以下の項で説明していきます．

これからの解説は，使われているCPUを限定しなければなりません．ここでは最も一般的な8ビットのマイクロプロセッサか，16ビットのインテル系のマイクロプロセッサ(8086，V30など)を使っていると仮定して話を進めます．

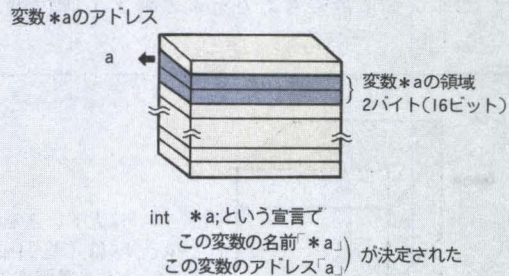
■C言語におけるポインタ

C言語の変数の宣言については，第2章で説明をしました．ここでは，int型で変数の宣言をした場合の変数のアドレスを例にします．

リスト 4-5 は，〝a〞という変数のアドレスを表示するプログラムです．またリストの下に，メモリ上で変数がどのように取られるのかを図示しています．

リスト 4-5　変数のアドレスを調べる

このプログラムでは，6行目の「&a」という表現で〝a〞のアドレスを求めています．つまり，変数名の前に「&」を付けることによって，その変数のアドレスを知ることができます．このとき変数aのアドレスを入れる変数addressは，かならずそのCPUに依存した大きさで宣言します．たとえば8086CPUのスモール・メモリモデルの場合には，16ビットでアドレスを表現しますからint型で宣言しておきます．また，CPUのアドレスには負の数はありませんから，unsigned型になります．そうしないと，アドレス計算などをするときに正しい計算結果が得られません．

さて，次に登場するのがC言語の特徴の1つであるポインタ変数です．まずはポインタ変数を使ったプログラムを見てみましょう(**リスト 4-6**)．このプログラムでは**リスト 4-5** とまったく同じことをやっています．

リスト 4-6　ポインタ変数を使ったプログラム

このプログラムについて，以下に解説します．

ある変数をポインタ変数として宣言する，たとえば"a"という名前のint型のポインタ変数を宣言するには，次のように行います．

```
int  *a;
```

この宣言によって，アドレス「a」に存在する「*a」という名前の16ビット長の変数が作られます．**リスト 4-5**で示した，

```
int  a;
```

という変数の宣言に比べると，頭に「*」が付いている点が異なります．しかし，こうすることによって今までの普通の変数とは違ったメリットが出てきます．通常の変数の宣言では，以下のようにその変数のアドレスを決めることはできません．

```
int  a;
&a = 0x0023;  ……  これはできない．エラーになる
```

ところがポインタ変数の場合は，こんな表現ができます．

```
int  *a;
a = 0x1C11;  ……  ポインタ変数に自由に代入できる
```

ポインタ変数で宣言した場合，その変数のアドレスは「&a」ではなく「a」で表します．そして，「a」にアドレスを入れることによって，変数の実体である「*a」は任意のメモリを参照することができます．

以下にポインタ変数の考え方をまとめます(**図 4-8**).

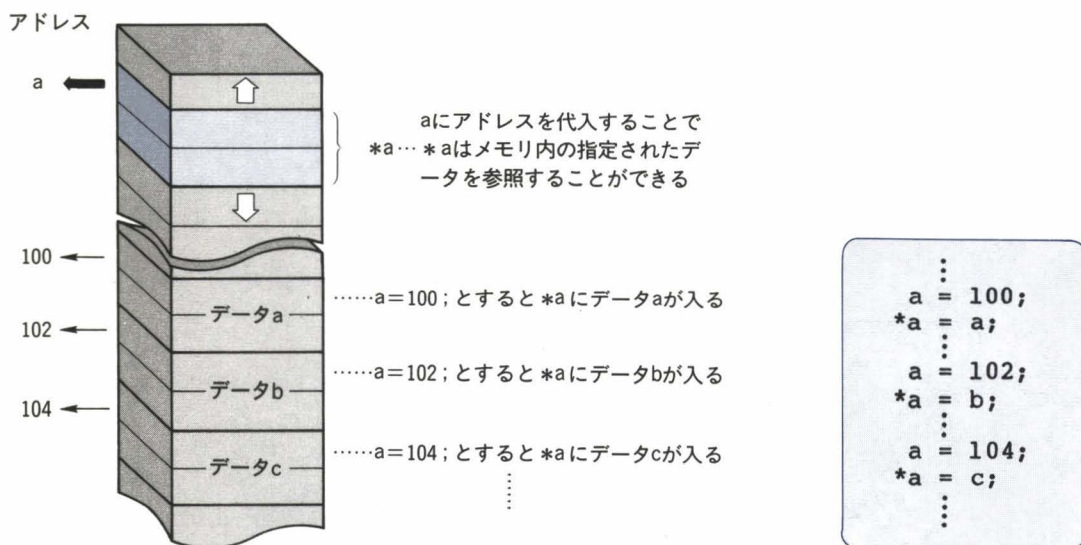



図 4-8　ポインタ変数の考え方

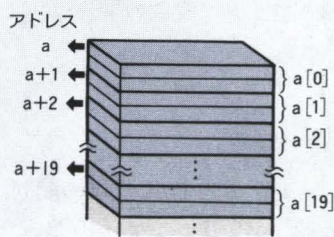
このようにポインタ変数とは，一言でいえば**アドレスを管理するための変数**です．また，ポインタ変数といった場合，どうも日本語のニュアンスでは〝ポインタの変数〃のように聞こえますが，〝ポインタ(アドレス)の操作が可能な変数〃と考えた方がよいでしょう．C言語の変数は，ポインタという観点からみると，以下の2種類に分けることができます．

C 言語の変数
- **通常の変数(ポインタの操作ができない変数)**
- **ポインタ変数(ポインタの操作ができる変数)**

ポインタ変数の特徴は，次に示す配列を扱った場合などにとくに威力を発揮します．

■配列とポインタ変数の込み入った関係

第2章で解説した配列は，ポインタ変数と密接なつながりを持っています．つながりというよりも，配列とポインタ変数は実は同一人物(？)だったのです．

まず，配列を使ったプログラムを見てみましょう(**リスト 4-7**)．

リスト 4-7 配列のアドレス

このプログラムは，〝a〞と言う名前のint型の20個の配列を宣言し，配列の先頭アドレスを求めるプログラムです．ここでわかるように，配列を宣言するとその配列名aは，そのままポインタ変数となります．つまり，

```
int a[20];
```

という宣言をすると，ここに使われている〝a〞という変数名は，

```
int *a;
```

と宣言したときと同じように変数のアドレスを示すことになります．配列の宣言では，int型の変数20個分のメモリが確保されますが，ポインタで宣言すると1つのメモリが確保されるだけです．逆に言えば，1つの配列を宣言した場合は，ポインタ変数で宣言するのと同じことになります．

```
int  a[1];        ……… 配列として宣言したとき
int  *a;          ……… ポインタ変数として宣言したとき
```

そして，この変数に格納した値は，

```
printf("Value is %d ¥n", a[0]);    ……… 配列として表現した場合
printf("Value is %d ¥n", *a);      ……… ポインタ変数として表現した場合
```

として画面に表示できます．これらは最初の宣言を配列で行ったとしても，ポインタ変数で行ったとしても，どちらでも，まったく同じようにprintf文などで使えます．つまり，「int　a[1];」で宣言した変数は，a[0]に格納されている値を知りたい場合，a[0]という表現と「*a」という表現のどちらも使うことができます．この配列とポインタ変数の関係は，これらの立場が逆になった場合もまったく同じことがいえます．

■ 文字配列と数値配列

前節では数値の配列しか扱いませんでしたが，文字の配列はどうなるのでしょうか？　もう少しくわしく，ポインタと配列の関係を探ってみましょう．

第2章の説明でわかることは，〝1文字は8ビットで表される″ということでした．この8ビットという情報量は，ここで前提としているCPUの最小のアドレス単位であることは，4.1節で述べたとおりです．つまり，文字の場合は〝1文字が1バイトを使う″，あるいは表現を変えれば〝1文字が1つのアドレスを占有する″といえます．

これに対してint型の変数は16ビットの大きさを持つ変数ですから，2つのアドレスを占めることになります．また，longだとか，floatだとか，doubleなどの変数も同様に2つ以上のアドレスを占有します．

char型とint型をポインタ変数で宣言した場合のメモリ上の扱いを見てみましょう(**図4-9**)．

この図のように変数*aのアドレスaは，その変数が占めるバイト数にかかわらず，かならず占有するメモリの先頭を指すことがわかります．

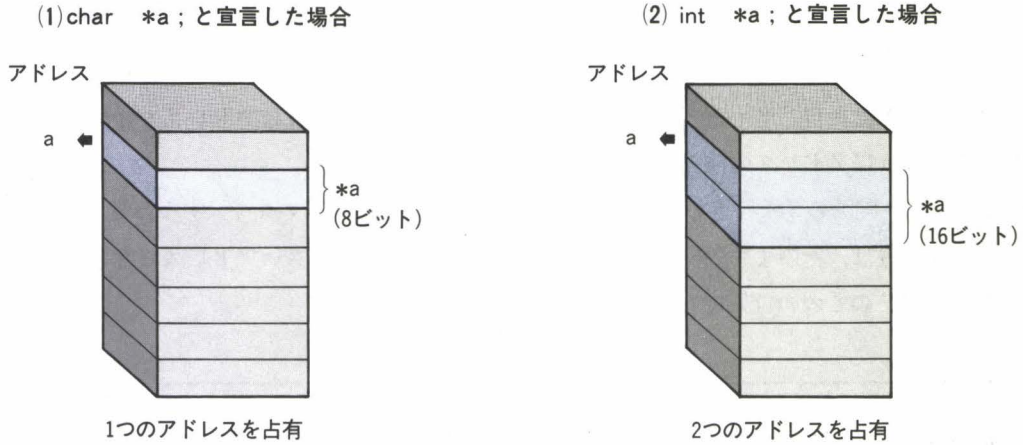



図 4-9 char 型と int 型のポインタ変数

さて次に,「char a[20] ;」として宣言された配列の中身を参照する方法を考えてみます. これには, 配列を使う場合とポインタを使う場合の2種類の方法を用いることができます(**表 4-2**).

参照する変数	配列の表現	ポインタの表現	実際のアドレス
1番目の数	a[0]	*a	A (注)
2番目の数	a[1]	*(a+1)	A+1
3番目の数	a[2]	*(a+2)	A+2
n番目の数	a[n−1]	*(a+n−1)	A+n−1
20番目の数	a[19]	*(a+19)	A+19

(注) Aの値は実行形式のファイルを作成したときに決定される

表 4-2 char 型変数の配列の参照

ここで，n番目の配列の中身を表すポインタの表現が，

＊(a＋n－1)

となっていることに注目してください．前述したように，ポインタ変数はアドレスを操作できる変数なので，このようにアドレスの演算が行えます．

表 4-2 の場合，実際のメモリ上のアドレスは**図 4-10** のようになっています．すなわち，(a＋3)のポインタの示す位置は，そのまま"a"から相対位置として3バイト分大きいアドレスに存在します．それはまた，a[3]そのものの位置でもあります．

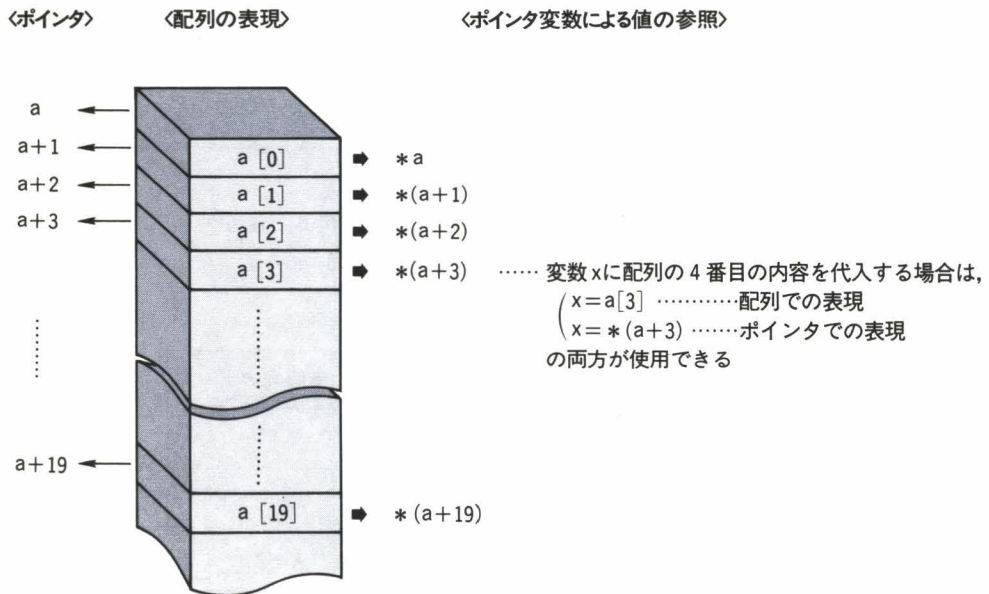



図 4-10　表4-2のメモリ上のアドレス

一方，「int　a[20]；」として配列を宣言した場合はどうなるのでしょうか．以下の**表 4-3** と**図 4-11** にその参照方法とメモリ上のアドレスを示します．

参照する変数	配列の表現	ポインタの表現	実際のアドレス
1番目の数	a[0]	*a	A(注)
2番目の数	a[1]	*(a+1)	A+2
3番目の数	a[2]	*(a+2)	A+4
n番目の数	a[n−1]	*(a+n−1)	A+(n−1)×2
20番目の数	a[19]	*(a+19)	A+19×2

（注）Aの値は実行形式のファイルを作成したときに決定される

表 4-3　int 型変数の配列の参照

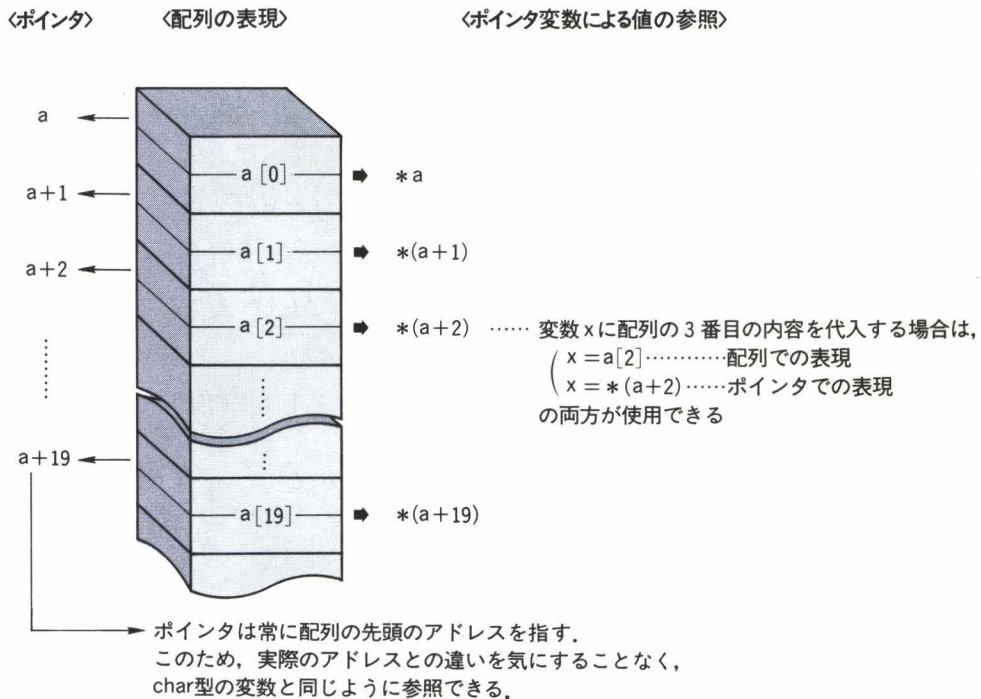



図 4-11　表4-3のメモリ上のアドレス

char型の変数と比較すると，ポインタは実際に参照するアドレスを1つおきに指しているのに，その表現は変わらないことがわかるでしょう．このような操作はコンパイラが自動的にやってくれるので，プログラマは〝ポインタと実際のアドレスのあいだの食い違いを気にしなくてもよい〟ということになります．これは重要なことです．たとえばプログラムを開発している最中に，今までint型で記述していた変数を必要に迫られてlong型に取って代えなくてはならないといったことがよくあります．C言語ではこんな場合も，最初の宣言でintをlongに代えるだけでほとんど用が足りてしまうのです．つまり，非常にマシン・アーキテクチャに密着した言語でありながら，マシンのアーキテクチャをあまり意識しないプログラミングが可能だということになります．

しかし，C言語で開発を行うプログラムは，主にシステムソフトが多いので，アセンブラ(機械語)との結合(リンク)が重要な要素になることがあります．この場合には，常に機械語を意識する必要があるので，ここでの説明が一概に当てはまるわけではありません．このことは，頭のすみに置いておいてください．

さて，以上でポインタの説明を終わります．ここで，アドレスとポインタという言葉が混乱してしまった人もいると思いますので，整理をしておきます．

アドレス　……　実際のメモリ上の位置
ポインタ　……　C言語でアドレスを操作しやすいように作った変数

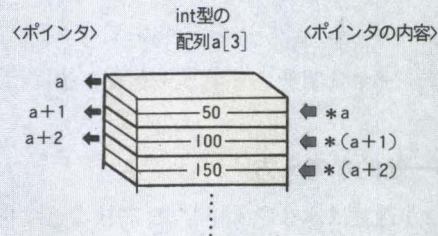
ここで学んだポインタ変数をいったいどのように使っていくのかは，これから先のプログラムで示していくことにします．以降のプログラムでポインタ変数についてわからなくなってしまったら，もう一度この節を読み返してください．

最後に，この節のまとめとしてポインタ変数を使った例題を**リスト4-8**に示しておきます．

```
 1: main()
 2:     {
 3:     int   a[3];
 4:     int   i;
 5:
 6:     a[0] = 50;
 7:     a[1] = 100;
 8:     a[2] = 150;
 9:
10:     for(i = 0 ; i < 3 ; ++i)
11:         {
12:         printf("a[%d] = %d ¥n", i, *(a + i));
13:         }
14:     }
```

[実行結果]

```
A>test
a[0] = 50
a[1] = 100
a[2] = 150

A>
```


MSX-C, LSI C は，1行目の前に「#include <stdio.h>」を追加

リスト 4-8　ポインタ変数を使ったサンプルプログラム

配列とポインタ変数の使い分け

これまでに述べたように，ポインタ変数と配列は自在に取り替えがききます．しかし，ポインタ変数で宣言されたものを理由もなしに配列として使ってみるということを，あちこちでやると，自分で自分の書いたプログラムがわからなくなってしまいます．C言語は，同じフローチャートで書ける同じ機能を持ったプログラムを非常に多くのバリエーションで書くことができる言語です．このバリエーションの豊富さはちょっと他の言語にはありません．しかし，このようなC言語の特徴は使い方しだいでよい結果を生む場合もあれば，そうでない場合もあります．この配列とポインタ変数の関係もその例の1つです．私はこのようなとき，〝配列で宣言したものは，特別な必要がないかぎりなるべく配列として扱う〞というごく平凡な原則を貫くことによって，これらの問題に対処しています．

4.4 関数

前節までで，C言語の大方の説明は終わりました．重要な項目でまだ解説していないことは，ここで説明する「関数」と「構造体／共用体」です．構造体，共用体はちょっとしたC言語のプログラムには必要がなく，また若干難解な説明をしなくてはいけないのでくわしい解説は本書では行いません（ちょっとだけ，次の章に顔を出します）．

C言語での関数はプログラミングの単位であり，大きな実行の単位です．他の言語でいえばサブルーチンに当たるものですが，BASICやFORTRANなどのサブルーチンとは根本的な考え方が違います．

ここでは，まず「関数」という言葉を正確につかむことから始めましょう．

■関数の一般的な考え方

関数という言葉はみなさん中学校，高校などで何度も聞いていると思います．人によっては，〝関数？数学についてはもうたくさんっ！〟という方も多いでしょう．しかし，関数という翻訳語には困ったものです．もともとの関数の英語訳である「function」という言葉を使った方が，ここでの説明には適当です．このfunctionは，関数という訳語のほかに，「機能」，「作用」，「職務」，「儀式」などの訳語も持っています．

これらをひっくるめて言い表せば，〝ある一定のものを与えると，一定の出力が得られるもの〟ということができます．たとえば自動販売機などは，この抽象化された関数の概念を説明するのによく使われます(**図4-12**)．

考えようによっては，世の中の出来事や物事はすべて〝ある物とある物のかかわりから生まれる出力〟であるといってもよいかもしれません．つまり，すべて「関数」なのです．

関数という概念の特徴の1つは，私たちがそれを使うのにその関数のなかで何が行われているのかをいっさい気にする必要がないということです．何が行われているかわからない部分のことを「ブラックボックス」といいます．このブラックボックスという考え方から，「関数」を「函数」（はこの数）といっている本もあります．

私たちはテレビの中身がどうなっているのかわかっていないし，それを使ってテレビ局で番組を作っている人たちも，そんなことは考えたこともない人がほとんどでしょう．しかし，私たちはテレビ

番組を選択し楽しむことができますし，テレビの中身を知らなくても番組の制作はできます．

このような世界の捉え方のことを，function(関数)といっているのです．

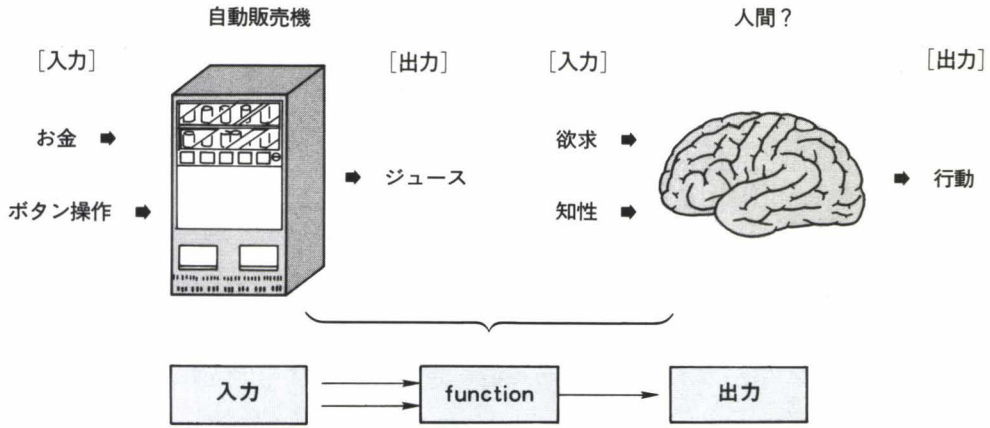



図 4-12 関数の概念

■関数を作る —その1—

C言語でいう「関数」も同じように，あるものを関数に入力するとそこで処理された結果が出力されます．第2章以来，ずっとお世話になっている「printf」も関数の1つです．printfの場合は，文字列とその出力形式を関数に渡すと，その結果として画面への出力が行われます．このあたりの事情をもっとよくわかってもらうために具体的な例を掲げて考えてみましょう．

ここでは,「printg」という関数を作ってみることにします(この名称はprintfの次という意味で付けました)．そのためには，まず始めにこの関数に入力されるものと出力されるものを，きちんと決めておく必要があります(**表 4-4**)．

入力	C言語での一連の文字列 例：static char a[11]="C Language";
出力	現在カーソルのある次の行の最初から,入力された文字列を表示し,表示後改行する

表 4-4 printg関数の仕様

そして，この関数は**リスト 4-9** のように使われます．

```
1: main()
2:      {
3:      static char a[35] = "** This is test today-is-fine **";
4:                      └──→ 配列a[ ]に文字列を用意しておく
5:      printf("%s¥n", a); ……… printf関数を呼び出す
6:
7:      printg(a); ………printg関数を呼び出す
8:      }
9:          ⋮
```

リスト 4-9　printg 関数の使われ方

printf は汎用の関数ですから，文字の表示を行う場合はそれを指定しなければなりません．また，printf 関数自身は改行を行わないので，表示する文字列のなかに「¥n」を書くのを忘れがちです．そこで，「1行書いたらかならず改行する」という機能を持った文字列表示用の関数を作ってみようというわけです．

さて，**リスト 4-9** を見てわかるように，この printg 関数の使用法は，printf 関数と非常に似かよっています．また，このプログラム上では，printf も printg もまったく同じ扱いを受けています．つまり，この関数も，その使い方を覚えてしまえば printf となんら変わらずに使用できるのです．これは BASIC や FORTRAN などではありえなかったことです．BASIC の場合，自分で PRINT 文を作ったり勝手にコマンドを直したりすることは，いっさいできませんでした．ところがC言語では，それがいとも簡単にできてしまいます．というのは，C言語で使われている printf などは，私たちが関数を書くのとまったく同じ方法で書かれているからです．C言語は，C言語自身を作った人たちにはできてもそれを使う人にはできないという〝差別〟が非常に少ない言語です．

では，printg 関数の実体を見てみましょう．

```
1: printg(a)
2:      char    a[];  } 変数の受け取り方や宣言の仕方については後述
3:      {
4:      printf("¥n%s¥n", a); ………配列a[ ]で渡された文字列を表示して改行する
5:      }
```

リスト 4-10　printg 関数

新しい関数を作るといったのに，4行目にはprintf関数が登場してきました．しかし，これは少しもおかしいことではありません．もともとprintf自身，もっと多くの〝他人の作った関数〟の集まりなのですから(printfという関数がいったいどこで定義されているのかということは，6章で明らかになります)．

また，1行目～2行目は初めてみる書式です．これについては，後でくわしく取り上げます．

さて，このprintg関数を使用したプログラムを組んでみましょう(**リスト 4-11**)．

```
 1: main()
 2:     {
 3:     char    a[20];
 4:     char    b[20];
 5:
 6:     printf("input Line-1    : ");  ……1行目のプロンプト文
 7:     scanf("%s", a);  ……1行目の入力
 8:
 9:     printf("input Line-2    : ");  ……2行目のプロンプト文
10:     scanf("%s", b);  ……2行目の入力
11:
12:     printg(a);  } printg関数を使って入力された文字列を表示
13:     printg(b);  }
14:     }
15:
16: printg(c)  ……printg関数の定義
17:     char    c[];  ……配列の宣言
18:     {
19:     printf("¥n%s¥n", c);  ……printf関数を使って,printg関数を実現している
20:     }
```

[実行結果]

```
A>test↵
input Line-1    : CP/M-80↵
input Line-2    : MS-DOS ↵

CP/M-80

MS-DOS
 A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト 4-11 printg関数を使ったプログラム

このようなC言語での関数は，他の言語でいえば〝サブルーチン〟に当たるというのはわかると思います．しかし，C言語での関数がサブルーチンと一線を画すのは，前述したように元から備わっている関数もプログラマが組んだ関数もまったく同様に扱えることにあります．

■関数を作る　―その2―

関数とサブルーチンには，もう1つ決定的な違いがあります．それは関数が値の受け渡しを行えるという点です[†]．値の受け渡しとはいってみれば入力と出力のことですから，これは関数の定義そのものでもあります．

BASICのサブルーチンは，まったく同じ作業を1カ所にまとめたものです．そこには，入力とか出力の概念はなく，サブルーチンが呼ばれたときは常に同じ結果が返ります．これに対して，関数は作業の本質をまとめたものといえます．つまり，BASICでは〝5＋3を計算する〟というサブルーチンしか作れなかったものが，C言語では〝任意の2つの値を加算する〟という関数を作成することができるのです．このため関数ではその利用範囲が広がり，その場限りでなく汎用的に使うことができます．

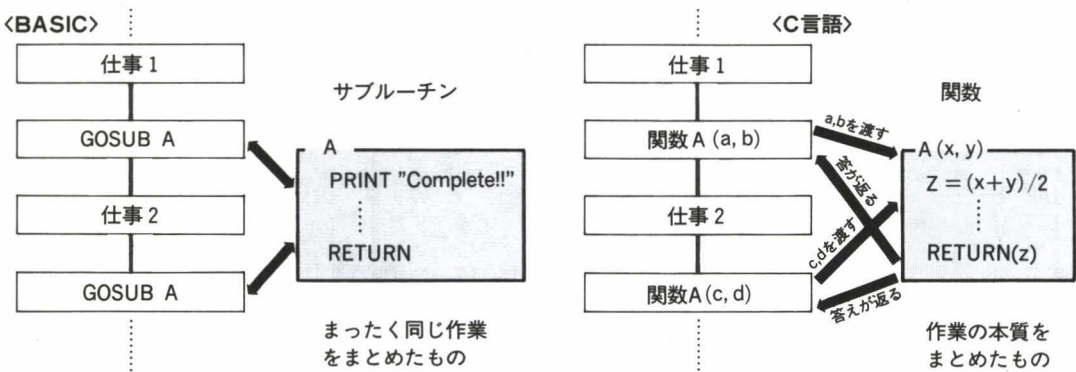



図 4-13　サブルーチンと関数

関数の本質がよくわかるように，もう1つプログラムを作ってみましょう．さきほどと同じように，まず関数の仕様を表にしておきます(**表 4-5**)．

入力	int型の変数aとb
出力	int型の変数c．cはaとbを加えた値である

表 4-5　b_and_a関数の仕様

† BASICの種類によっては，サブルーチンで値の受け渡しを行えるものもあります．ここでのBASICとは，代表的なMicrosoft系のBASICを指しています．

この関数は，「 b_and_a 」という名前で2つの変数の加算を行い答えを返します．**リスト 4-12** にこの関数を使ったプログラムを示します．

```
 1: main()
 2:      {
 3:      int   a, b, c;
 4:
 5:      printf("Number <a> : ");
 6:      scanf("%d", &a);
 7:
 8:      printf("Number <b> : ");
 9:      scanf("%d", &b);
10:
11:      c = b_and_a(a, b); ………入力された値を関数b_and_aに渡し，その結果を変数cに
12:                                受け取る
13:      printf(" b and a = %d ¥n", c);
14:      }                                          b_and_a関数の本体
15:
16: b_and_a(x, y)  }
17:      int x, y; } 変数の受け取り方については後述
18:      {
19:      int z;
20:
21:      z = x + y; ………合計の計算
22:
23:      return(z); ………計算した値を返す場合には
24:      }                 return文を使う
```

[実行結果]

```
A>test ↵
Number <a> : 300 ↵
Number <b> : 987 ↵

b and a = 1287
A>
```

MSX-C，LSI C は，1行目の前に「#include　<stdio.h>」を追加

リスト 4-12　b_and_a 関数を使ったプログラム

このプログラムの8行目で，

c = b_and_a(a, b);　……　変数 a と b を b_and_a 関数に渡し，その結果を変数 c に受け取る

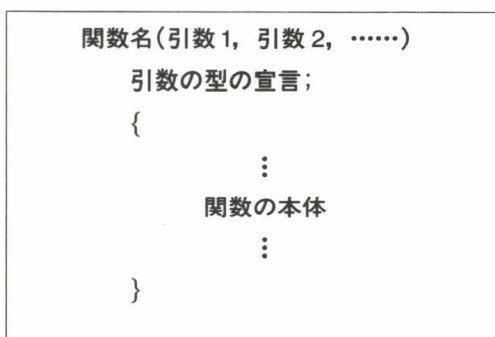
という形で値の受け渡しが行われています．ここで，変数 a と b にどんな値が入っているかによって返ってくる値が異なることになります．実際の関数のなかではもっと複雑なことを行う場合が多いのですが，その場合も同じように〝入力と出力〟という形で関数が利用できます．

C言語では，関数間の値の入力と出力にいくつかの決まったパターンがあります．それを次に解説します．

■関数間のデータの受け渡し

C言語のプログラムは，関数の集まりとして記述されるので，各関数の間でどのようにデータの受け渡し(入力と出力)を行うのかが重要な問題となります．ここでは，関数間のデータの受け渡しについてまとめておくことにします．

まず，関数の書式を以下に示します．関数間で渡されるデータ(変数)は**引数**と呼ばれます．引数は，関数名に続く「( )」のなかに書かれ，その実行単位が始まる前に宣言されます．また，引数は省略されることもあります．

```
関数名(引数1, 引数2, ……)
    引数の型の宣言;
    {
            ⋮
        関数の本体
            ⋮
    }
```

関数間の値の受け渡しは，いくつかの場合に分類して考えることができます．まず，関数には，「呼ぶ側」と「呼ばれる側」の2種類があります(**図 4-14**)．そのため，値を渡す方向として以下の場合が考えられます．

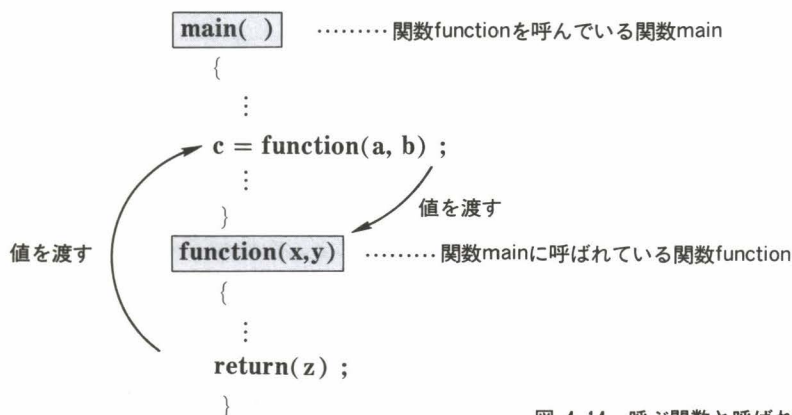



図 4-14　呼ぶ関数と呼ばれる関数

1．呼ぶ側から呼ばれる側に値を渡す場合
2．呼ばれる側から呼ぶ側に値を渡す場合

また，値を渡す方法としては以下の4つが考えられます．

1．関数の引数として変数を直接渡す場合
2．関数の引数として変数のポインタを渡す場合
3．関数の返値(関数値：return value)として渡す場合
4．関数の外に設定した変数(グローバル変数)を使って値を渡す場合

これらの組み合わせを考えると2×4で合計8通りに分かれますが，そのすべてが利用できるわけではありません．このうちC言語のプログラムで使えるものを**表4-6**に示します．

値を受け渡す方向	変数の値そのもの	変数のポインタ	関数の返値	グローバル変数
〈呼ぶ側〉➡〈呼ばれる側〉	○	○	×	○
〈呼ばれる側〉➡〈呼ぶ側〉	×	○	○	○

表 4-6 関数間の値の受け渡し一覧

この1つ1つについて，サンプルプログラムを挙げながら以下に解説していきます．

◆関数の引数として変数を直接渡す場合[呼ぶ側→呼ばれる側]

```
 1: main()
 2:      {
 3:      int a, b;
 4:
 5:      a = 2; b = 3;
 6:
 7:      fx1(a, b); ……… 変数a, bの値が関数fx1の変数ax, bxに渡される
 8:      }
 9:
10: fx1(ax, bx) ———— 引数の変数名は，呼ぶ側と同じである必要はない
11:      int ax;   引数の宣言 { ・関数の実行単位が始まる前に宣言する
12:      int bx;             { ・呼ぶ側のデータ型と一致している必要がある
13:      {
14:      int cx;
```

```
15:
16:      cx = ax + bx; ………axとbxを使って計算ができる
17:
18:      printf("<fxl> ax + bx = %d¥n", cx);
19:      }
```

［実行結果］

```
A>test⏎
<fxl> ax + bx = 5
A>
```

MSX-C，LSI C は，1行目の前に「#include　<stdio.h>」を追加

リスト 4-13　引数として変数を渡すサンプルプログラム

この場合は，引数として渡した値がそのまま呼んだ先の関数の引数にコピーされます(**図 4-15**)．つまり，**呼ばれた側でその変数の値を変更しても呼ぶ側の変数は影響がありません．**

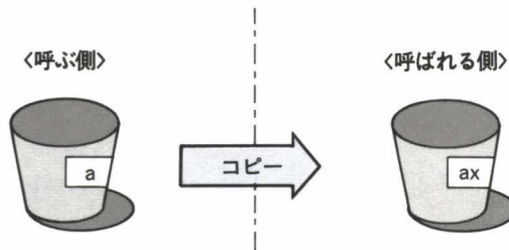
この方法は，一方通行で「呼ぶ側→呼ばれる側」には値を渡せますが「呼ぶ側←呼ばれる側」へ値を渡すことはできません．また，配列を渡すこともできません．しかし，引数として入れた値が直接渡るので，わかりやすいプログラムとなります．

変数 a が変数 ax にコピーされるので変数
ax の値を書き変えても変数 a は影響がない

図 4-15　変数を直接渡す場合

◆関数の引数として変数のポインタを渡す場合[呼ぶ側→呼ばれる側]

<変数のポインタを渡す場合>

```
 1: main()
 2:     {
 3:     int a, b;
 4:
 5:     a = 2; b = 3;
 6:
 7:     fxl(&a, &b);  ………引数として変数のポインタを渡している
 8:     }
 9:
10: fxl(ax, bx)
11:     int *ax;  }ポインタ変数として宣言する
12:     int *bx;  }
13:     {
14:     int cx;
15:
16:     cx = *ax + *bx;  ………*axと*bxを使って計算ができる
17:
18:     printf("<fxl> *ax + *bx = %d¥n", cx);
19:     }
```

[実行結果]

```
A>test↵
<fxl> *ax + *bx = 5
A>
```

<配列のポインタを渡す場合>

```
 1: main()
 2:     {
 3:     char    a[4];
 4:
 5:     fxl(a);  ………引数として配列の先頭のポインタを渡している
 6:     }
 7:
 8: fxl(ax)
 9:     char    ax[];  ………配列として宣言する
10:     {
11:
12:     ax[0] = 'A';  }
13:     ax[1] = 'B';  }ax[ ]を使った処理ができる
14:     ax[2] = 'C';  }
15:     ax[3] = 0;    }
16:
17:     printf("<fxl> ax string : %s¥n", ax);
18:     }
```

[実行結果]

```
A>test↵
<fxl> ax string : ABC
A>
```

MSX-C, LSI C は, 1行目の前に「#include <stdio.h>」を追加

リスト 4-14　引数として変数のポインタを渡すサンプルプログラム(1)

◆関数の引数として変数のポインタを渡す場合[呼ぶ側←呼ばれる側]

〈変数のポインタを渡す場合〉

```
 1: main()
 2:      {
 3:      int a, b;
 4:
 5:      fx1(&a, &b); ………関数fx1を呼び出す
 6:
 7:      printf("<main> a + b = %d¥n", a + b);
 8:      }
 9:
10: fx1(ax, bx)
11:      int *ax;
12:      int *bx;
13:      {
14:      *ax = 2;
15:      *bx = 3;
16:      }
```

関数fx1を呼び出した後は
変数*axと*bxの内容が
関数mainのa,bに入る

[実行結果]

```
A>test
<main> a + b = 5
A>
```

〈配列のポインタを渡す場合〉

```
 1: main()
 2:      {
 3:      char    a[4];
 4:
 5:      fx1(a); ………関数fx1を呼び出す
 6:
 7:      printf("<main> a string : %s¥n", a);
 8:      }
 9:
10: fx1(ax)
11:      char    ax[];
12:      {
13:      ax[0] = 'A';
14:      ax[1] = 'B';
15:      ax[2] = 'C';
16:      ax[3] = 0;
17:      }
```

関数fx1を呼び出した後は
配列ax[ ]が
関数mainのa[4]に入る

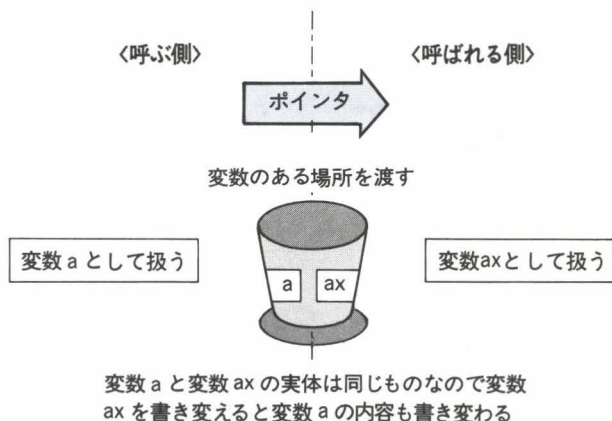
[実行結果]

```
A>test
<main> a string : ABC
A>
```

MSX-C, LSI Cは，1行目の前に「#include ＜stdio.h＞」を追加

リスト 4-15　引数として変数のポインタを渡すサンプルプログラム(2)

この２つの場合は，変数の値ではなく変数のポインタが渡されることが特徴です．ポインタはその変数のメモリ上のアドレスを示すので，値のコピーではなく値のある場所が渡されると考えることができます(**図4-16**)．つまり，呼ばれる側でその値を書き変えてしまうと，呼ぶ側の値も同じように変わってしまいます．

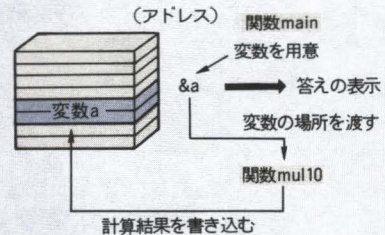
図 4-16 変数のポインタを渡す場合

ポインタを使うと，配列を渡すことも可能になります．また，呼ぶ側と呼ばれる側のどちらから値を渡す場合でも，その書式はまったく同じです(**リスト 4-14** と **4-15** を見比べてください)．つまり，値の渡す方向は利用するときに決めればよく，同時に双方向の値の受け渡しもできます．この例題を１つ以下に示します(**リスト 4-16**)．

```
 1: main()
 2:      {
 3:      int a;
 4:
 5:      printf("Number = : ");
 6:      scanf("%d", &a);
 7:
 8:      mul10(&a);  ………入力された数値が入っている
                       変数aのポインタを渡す
 9:
10:      printf("Number * 10 = : %d¥n", a);
11:      }
12:
13: mul10(ax)
14:      int *ax;  ………ポインタ変数で受け取る
15:      {
16:      *ax *= 10;   計算結果を変数aに返す
17:      }
```

（アドレス）　関数main　変数を用意　&a　答えの表示　変数a　変数の場所を渡す　関数mul10　計算結果を書き込む

［実行結果］

```
A>test↵
Number = : 10↵
Number * 10 = : 100
A>
```

MSX-C, LSI C は，1行目の前に「#include　<stdio.h>」を追加

リスト 4-16　2つの関数で同時に値を引き渡すサンプルプログラム(1)

◆関数の返値（関数値）として値を渡す場合［呼ぶ側←呼ばれる側］

```
 1: main()
 2:      {
 3:      int a;
 4:
 5:      a = fx1();  ………関数fx1を呼び出し，その返値を変数aに受け取る
 6:
 7:      printf("<main> a : %d¥n", a);
 8:      }      ………引数はなし
 9:
10: int fx1()  ………関数が返値として渡す型を宣言（ここではint型の数値を返す）
11:      {
12:      return(-1);  ………関数mainに返値を渡す
13:      }
```

［実行結果］

```
A>test ↵
<main> a : -1
A>
```

MSX-C, LSI C は，1行目の前に「#include　<stdio.h>」を追加

リスト 4-17　関数の返値を渡すサンプルプログラム

リスト4-17では単に値を返すだけですが，**リスト4-14，4-15，4-16**のように[呼ぶ側→呼ばれる側]に引数を渡すこともできます．その場合変数を渡す方法は，直接渡してもポインタを渡してもかまいません．つまり，これまで説明した方法を使っても双方向の受け渡しができます(**リスト4-18**)．ただし，返値として渡せるのは1つの値だけです．もし，2つ以上の値を渡したい場合は，前述のポインタによる方法を用いる必要があります．

```
 1: main()
 2:     {
 3:     int a, b;
 4:
 5:     printf("Number = : ");
 6:     scanf("%d", &a);
 7:
 8:     b = mul10(&a);   ………関数mul10に変数aのポインタを渡し，その返値を変数bに受け取る
 9:
10:     printf("Number * 10 = : %d¥n", b);
11:     }
12:
13: int mul10(ax)   ………関数の返値はint型
14:     int *ax;   ………引数をポインタ変数として受け取る
15:     {
16:     *ax *= 10;
17:
18:     return(*ax);   ………関数の返値はreturn文で返す
19:     }
```

[実行結果]

```
A>test↵
Number = : 50↵
Number * 10 = : 500
A>
```

MSX-C，LSI Cは，1行目の前に「#include <stdio.h>」を追加

リスト 4-18　2つの関数で同時に値を引き渡すサンプルプログラム(2)

値を返値として渡す場合には，呼ばれた側でreturn文を使います．return文の書式は，以下のとおりです．

return(関数値);

ここで関数値は通常の変数と同じように，intやcharなどのデータの型を持っています．そこで，その関数の返す返値を明確にするために，関数を定義する最初に以下のように宣言をしておきます．

```
long  func1(a, b)
  long    a;
  long    b;
  {
          ⋮
  return([long の値]);
  }
```
この関数が返す返値は，long 型である

◆グローバル変数を使って値を渡す場合[呼ぶ側⇄呼ばれる側]

```
 1: int a; ………………グローバル変数として変数aを宣言
 2:
 3: main()
 4:      {
 5:      a = 3;
 6:
 7:      fx1(); ………関数fx1を呼ぶ
 8:
 9:      printf("<main> a : %d¥n", a);
10:      }
11:
12: fx1()
13:      {
14:
15:      a += 2; ……この変数は関数mainと同じもの
16:
17:      }
```

[実行結果]

```
A>test↵
<main> a : 5
A>
```

MSX-C，LSI C は，1 行目の前に「#include　<stdio.h>」を追加

リスト 4-19　グローバル変数を使ったサンプルプログラム

関数の外で宣言されたグローバル変数は，すべての関数でその変数を参照できます．これを利用すれば，**リスト 4-19** のように関数間で値の受け渡しが行えます．しかし，この方法ですべてのプログラムを書いてしまうのは感心しません．

■関数のまとめ

よく言われるように，関数の概念はC言語の要です．もう一度，まとめの意味で関数の一般的な型を見てみましょう(**図 4-17**)．

```
func(a, b, c) ………まず,関数名と引数が書かれる

    int     a;  ┐
    long    b;  ├ 引数のデータ型の宣言
    char    c;  ┘

    {           ………関数の実行単位の始まり
    int     i; ……実行単位中で使われる引数以外の変数を宣言する
     ⋮          } 実行単位
    }       ……………… 実行単位の終わり
```

図 4-17 関数の一般形

図 4-17 は，第 3 章で学んだ制御構造とそっくりです．制御構造の書式も以下に示してみます(**表4-7**)．

種類		書式
制御構造	for文	for(制御文){実行文;実行文;……}
	if文	if(制御文){実行文;実行文;……} else ＊{実行文;実行文;……} ＊elseの制御文はifの制御文と同じなので省略してあると考えられる
	while文	while(制御文){実行文;実行文;……}
	switch〜case文	switch(制御文){case:実行文; case:実行文……}

表 4-7 制御構造の書式

図 4-17 と**表 4-7** からわかるように，C言語の意味のある構文は，かならず，

名前(入力される値) 引数の宣言 {実行文;実行文;…… }

という形をしています．考えようによっては，for，if などの制御文も予約された一種の関数であると言えます．そして，この「{実行文； ……}」のなかに，また違う関数を使うことができるという構造をしているわけです．これが，C言語のプログラムがすべて関数で記述できる理由なのです．

【この章のポイント】

1．コンピュータのメモリの構造
2．C言語のメモリの使い方には，基本的に「auto」と「static」がある
3．ポインタは，アドレスを管理するための変数である
3．ポインタ変数と配列は，同じものとして扱える
4．ポインタは，四則演算ができる．また宣言した変数の大きさによって，コンパイラが操作するので，実際のアドレスとの食い違いを気にする必要はない
5．C言語のプログラムは，関数の集まりである
6．関数間の数値の渡し方

［練習問題］

1．次のプログラムに書き加えて，変数aの値を変数bに代入するプログラムを作りなさい(作ったら実際に試してみてください)

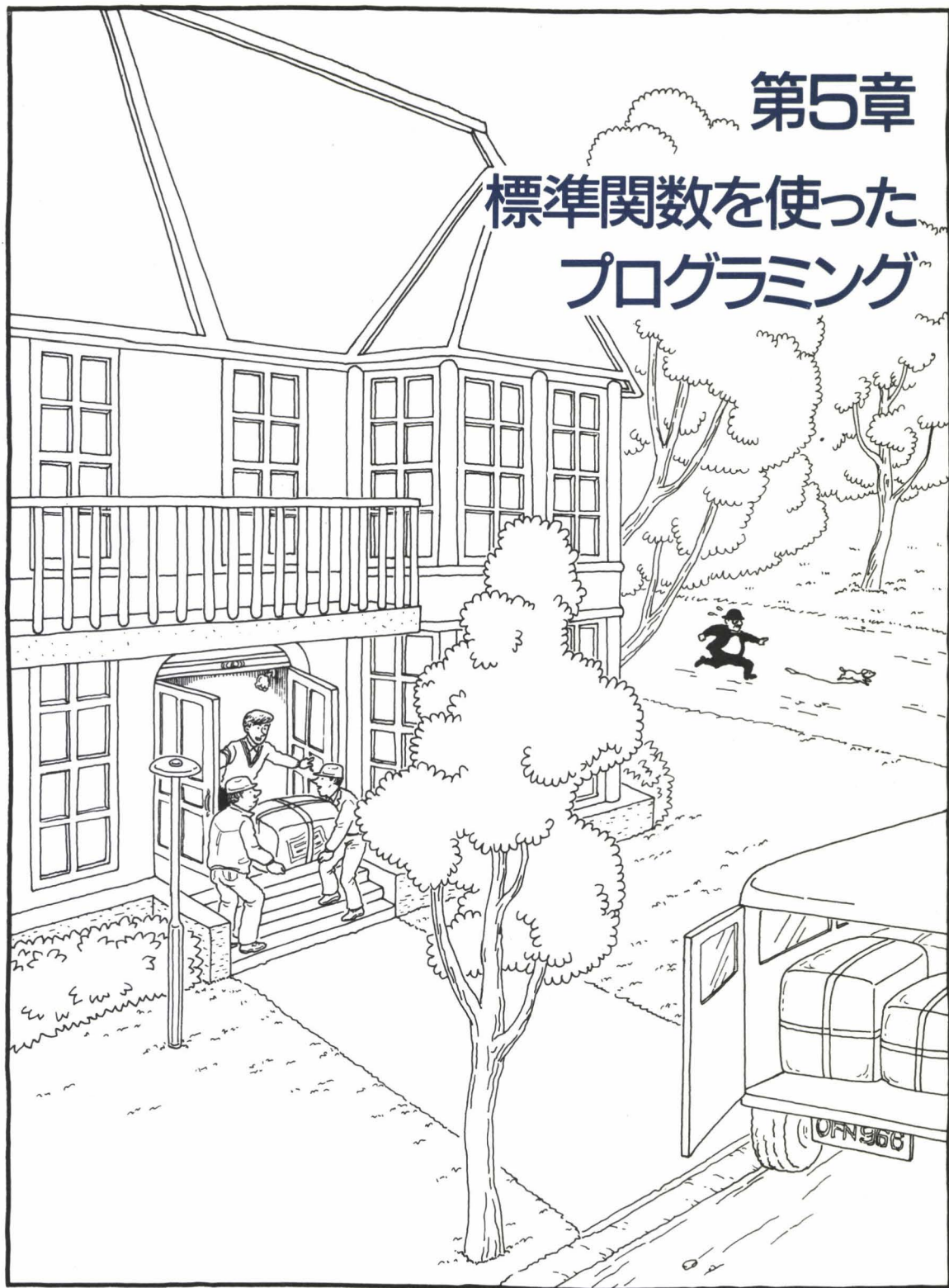
```
main()
    {
    long     a, *b;

    a = 3250;
              :
     (この間を作成する)
              :
    printf(" a == b == %d¥n", *b);
    }
```

2．あなたの使っているパソコンのある場所のメモリを読んで，16進数で画面に表示するプログラムを作りなさい

3．BASICのINPUT文と同じ働きをする関数を書きなさい(何種類作ってもかまいません)
その際，めんどうでも「仕様(入出力)の表」と「その関数を使うプログラム」も作ること

（解答は218ページ）

第5章
標準関数を使ったプログラミング

C言語は，それ自身にファイル入出力などの機能があるわけではなく，制御構造や演算子のみを扱うコンパクトな言語です．そのかわりC言語には，入出力やファイル操作などの多くの関数が別に用意されています．この関数のことを標準関数(ライブラリ関数)と呼びます．

前章でも述べたように，C言語のプログラムは関数の集まりとして記述されるので，関数をどのように組み合わせてプログラムを作っていくのかが重要なポイントです．一言でいってしまえば，C言語のプログラミングとは，標準関数を使って自分専用の関数をどんどん定義していくことです．たとえば，特定のエラーが起きた場合の対策を含めたファイルのオープン用の関数や，あるファイルの一部をいつも参照するための関数などを作っていきます．

この章では，これらの基礎となる標準関数について説明します．標準関数は，各社のC言語によって付いてくるものやその使用法が異なるものもありますが，ここではそのなかでも一般的な関数を紹介します．ここで関数という以上はどこかにその実体が定義されているわけですが，これについては次章で取り上げることにします．

また，標準関数を使ったプログラムを組んでいく際の参考となるように，最も基本的なファイル入出力を行うプログラム例を挙げます．

5.1 システム標準関数

C言語には，コンパイラのほかに「システム標準関数」が一式付いてきます．これらは，BASICのコマンドと同じようにその使用法や機能がきちんと決まっています．そして前章でも述べたように，〝値の受け渡し〟という形で利用することができます．おもな関数の細かな仕様はAPPENDIXに整理してありますので，ここでは各関数を機能別に分類しその概略をまとめておくことにします．

■キャラクタコード分類関数

この関数群は，与えられた文字（1文字）の種類を調べます．たとえば，〝アルファベットと数字〟や〝目に見える文字と目に見えないコントロール文字〟などの分類を行います．これらの関数は，if文などで1つずつ処理をしているわけではなく，すべて「テーブル参照」(table look-up)という方法で行われているので非常に高速です．〝この文字は数字か否か〟などの判断には，この関数を使ってください．

表5-1にこれらの関数とその機能を掲げます．

isalnum(c)	アルファベットまたは数字か，そうでないかを調べる
isalpha(c)	アルファベットかどうかを調べる
isascii(c)	正しいアスキーコード(0x00～0x7F)かどうかを調べる
iscntrl(c)	制御文字(0x20より小さいコード)かどうかを調べる
isdigit(c)	値が0～9かどうかを調べる
islower(c)	小文字かどうかを調べる
isprint(c)	画面(あるいはプリンタ)に出力可能な文字かどうかを調べる
ispunct(c)	句読点(ピリオドなど)かどうかを調べる
isspace(c)	空白文字(タブやスペースなど)かどうかを調べる
isupper(c)	大文字かどうかを調べる

表 5-1 キャラクタ分類関数

■ 文字列操作関数

この関数群は，文字列の操作や認識を行います．処理の対象となる文字列は，すべて**Cストリング**でなければなりません．Cストリングについては2章でも説明しましたが，ある文字からキャラクタ0 (NULL コード) までの文字列を指します．

表 5-2 に文字列操作関数とその機能を示します．

strcat(s1, s2)	文字列をつなげる
strncat(s1, s2, n)	文字列をつなげる(ただし n 文字分しか行わない)
strcmp(s1, s2)	2つの文字列の比較を行う
strncmp(s1, s2, n)	2つの文字列の比較を行う(ただし n 文字までしか行わない)
strlen(s1)	文字列の長さを調べる
strcpy(s1, s2)	文字列のコピーをする
strncpy(s1, s2, n)	文字列のコピーをする(ただし n 文字までしか行わない)

表 5-2　文字列操作関数

■ ファイル操作関数

ファイル操作とは，ファイルのオープン/クローズやファイル関係のエラー処理，さらにファイルバッファの割り当てなど，ファイル入出力以外のあらゆるファイルの処理のことをいいます．

表 5-3 にこれらの関数とその機能を示します．

fopen(file, mode)	ファイルをオープンする
freopen(file, mode, fp)	指定したファイルをクローズしたのち，ファイルをオープンする
fclose(fp)	ファイルのクローズを行う
fflush(fp)	ファイルバッファ上のデータをファイルに書き出す
setbuf(fp, b)	ファイル入出力バッファの指定を変える
feof(fp)	ファイルが終わっているかどうかを調べる
ferror(fp)	ファイルがエラーかどうかを調べる
clearerr(fp)	ファイルのエラー状態を解除する
fileno(fp)	ファイルのコネクション番号を調べる

表 5-3　ファイル操作関数

■バイト入出力関数

ファイルやキーボード／ディスプレイなどへ1バイトずつの入出力を行う関数です．この後に出てくるワード単位とストリング単位の入出力関数は，すべてこの関数をもとに作られています．

このなかで，そう頻繁に使うことはないのですが，ちょっと変わった関数があります．それは，ungetc 関数で，これは〝入力関数なのに出力を行う〟のです．

表 5-4 にバイト入出力関数を示します．

fgetc(fp)	ファイルから1バイト読み込む
fputc(fp)	ファイルへ1バイト書き出す
getchar( )	標準入力から1バイト読み込む
putchar(c)	標準出力へ1バイト書き出す
getc(fp)	fgetcと同じ
putc(c, fp)	fputcと同じ
ungetc(c, fp)	読み込んだ1文字をファイルへ戻す

表 5-4　バイト入出力関数

■ワード入出力関数

前述の関数群がバイト入出力であるのに対して，これらはワード(2バイト)単位で入出力を行います．**表 5-5** にワード入出力関数を示します．

getw(fp)	ファイルから1ワード(2バイト)を読み込む
putw(i, fp)	ファイルへi(ワード)書き出す

表 5-5　ワード入出力関数

■ 文字列入出力関数

文字列の入出力を行います．先の文字列操作関数と同様に，Cストリングがその対象となります．**表 5-6** に文字列入出力関数を示します．

fgets(b, n, fp)	nバイトまでの1行をファイルから読み込む
gets(b)	標準入力から1行読み込む
fputs(b, fp)	ファイルへ1行書き出す
puts(b)	標準出力へ1行書き出す

表 5-6　文字列入出力関数

■ ブロック入出力関数

任意の長さのデータブロックごとに入出力を行う関数です．ランダムアクセスを行う場合やバイナリファイル(実行形式のファイルなど)の入出力によく使われます．

表 5-7 にこれらの関数とその機能を示します．

fread(b, s, n, fp)	ファイルからn個のsバイトのブロックを読み込む
fwrite(b, s, n, fp)	ファイルへn個のsバイトのブロックを書き出す

表 5-7　ブロック入出力関数

■ フォーマット化入出力関数

データをある一定の形式で入出力するための関数です．これまでよく出てきたprintfとscanfがその代表ですが，このほかに〝表現指示のための文字〟を使ってファイルなどと入出力を行う関数が加わっています．**表 5-8** にフォーマット化入出力関数を示します．

printf(format[, list])	標準出力に書式指定された文字列を書き出す
fprintf(fp, format[, list])	ファイルに書式指定された文字列を書き出す
sprintf(b, format[, list])	文字配列に書式指定された文字列を書き出す
scanf(format[, list])	標準入力から書式指定された値を読み込む
fscanf(fp, format[, list])	ファイルから書式指定された値を読み込む
sscanf(b, format[, list])	文字配列から書式指定された文字列を読み込む

[ ]は省略可能

表 5-8　フォーマット化入出力関数

■ランダムアクセス関数

ファイルをランダムアクセスするための関数です．ファイルのアクセスには，ファイル上の位置を表す「シークポインタ」(ファイルポインタ)を用います．このポインタを使うことによって，ファイル中のどこでも自由にアクセスすることが可能になります．

表 5-9 にランダムアクセスのための関数を示します．

fseek(fp, offset, mode)	ファイルポインタを操作する
ftell(fp)	現在のファイルポインタの値を調べる
rewind(fp)	ファイルポインタをファイルの最初に戻す

表 5-9 ランダムアクセス関数

■ソート関数

データをソート(並べ替え)するための関数もあります．しかし，キーとなるデータのみがソートされるので，それ以外に並べ替えたいデータがある場合はキーとともに移動してやる必要があります．

ソート関数を以下の**表 5-10** に示します．

qsort(b, n, s, p)	クイックソートを行う

表 5-10 ソート関数

■メモリ割り当て関数

プログラム中で一時的に大きなメモリを必要とした場合に，これらの関数を使うとメモリ領域を割り当ててくれます．たとえば，大きなファイルバッファや画像処理用の一時的な領域が欲しいというときに使われます．ただし，これらの関数は他の標準関数からも呼ばれていますので，使用に当たっては十分な注意が必要です．また，メモリエリアを使った後は，かならず free 関数を使って解放しておいてください．

なお，このメモリ割り当ては，ダイナミック・メモリ・アロケーション(動的メモリ配置)とも呼ばれます．**表 5-11** にメモリ割り当て関数を示します．

calloc(n, s)	n個の s バイトのメモリ領域を一時的に割り当てる
free(p)	割り当てたメモリ領域を解放する
malloc(s)	s バイトのメモリ領域を一時的に割り当てる

表 5-11 メモリ割り当て関数

■ その他の関数

その他というと，どうでもよいと思われるかもしれませんが，実はかなり使用頻度の高い関数があります．とくに文字列を変換する「atoi/atol/atof」などはよく使います(**表 5-12**)．

exit(s)	実行を終了し，オペレーティング・システムに戻る
abs(i)	絶対値を求める
atof(s)	文字列で表現された値を浮動小数点数に変換する
atoi(s)	文字列で表現された値を整数に変換する
atol(s)	文字列で表現された値をlong型の整数に変換する
rand( )	乱数を発生する
srand(seed)	乱数発生関数の基数を変える
setjmp(env)	スタックなどの環境を保存する
longjmp(env, value)	setjmpで保存された環境を受け取る

表 5-12　その他の関数

標準関数を使う際の注意

これまで見てきた標準関数は，多くのＣ言語で一般に備わっているものばかりです．このほかＣ言語によっては，より便利な関数が用意され，それを売りものにしているものもあります．

しかし，だからといってあまり特殊な関数ばかり使っていると後で困ることが出てきます．それは，「互換性の問題」です．これまで何度も書いたようにＣ言語の使われる理由の１つは，ハードウェアやＣ言語の種類に依存しないプログラムが組めるからです．〝あるＣ言語では動くが別のＣ言語では動かない〟とか〝あるコンピュータでは動くが別のコンピュータでは動かない〟というようなプログラムを書くことはできるだけ避けるべきです．また移植性を考慮しなければ，Ｃ言語を使うよりもFORTRANや機械語で組んだほうがよい場合もあります．

ハードウェアの進歩が著しい今日では，使っているマシンがいつ時代遅れになるかわかりません．そのためにも今作っているプログラムが他のハードウェアでも動くように，考慮しておきたいものです．Ｃ言語は互換性のあるプログラムを書くことができる言語ですから，とくに初心者のうちはここで紹介した「どこのＣ言語にも，同じ形で用意されている」標準関数を使ったプログラミングを心がけましょう．

5.2 ファイルの入出力 ―入出力関数を使ったプログラミング―

前節で説明した標準関数のなかでとくに頻繁に使われるのは，ファイルの入出力関数です．またファイルとのやりとりは，プログラムを組む上での基礎となる課題です．そこでこの節では，ファイルを扱うプログラムを作っていくことにします．また，標準関数がいくつか出てきますので，どのようにプログラム中で使われるのかを理解してください．

■C言語で扱うファイルとは

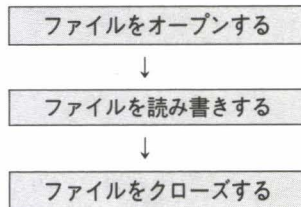
ファイルとは，〝あるまとまりを持ったデータの集まり〟のことです．この概念は，どのプログラミング言語でも変わるところはありません．また，ファイルを構成する要素とファイルを扱う手順も同じように考えることができます．それを以下にまとめておきます．

◆ファイルを構成する要素

・外部記憶装置(ディスクドライブ)上の一連のデータ列

・一連のデータに付けられたデータ番号

・それを管理するオペレーティング・システム

◆ファイルを扱う手順

ファイルは，単純に言うと〝メモリをディスク上に置き換えたもの〟と考えることができます．ファイル上のデータには，メモリと同じようにその１つ１つに番号が付いており，またデータを管理するためのポインタ(ファイルポインタ)を持っています．そして，この**ファイルポインタ**を使ってファイルのどこのデータも自由に読み書きができます．

ただしメモリと違うのは，ディスク上のどのファイルを扱うかを指定するためにオープン/クローズという作業を行う必要があることです．**図 5-1** にファイルの扱い方を示します．

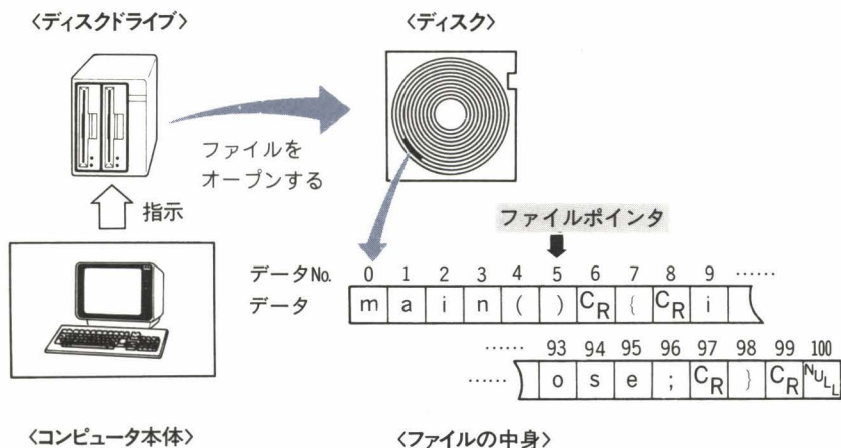



図 5-1　ファイルの概念とファイルポインタ

■オペレーティング・システムによるファイルの扱いの違い

C言語でファイルを扱う場合に，いくつか注意しなければならないことがあります．C言語はもともとUNIXというオペレーティング・システムから生まれてきた言語なので，UNIXの特徴をかなり引き継いでいます．そのためMS-DOSやCP/Mなどのオペレーティング・システムを使ってC言語のプログラムを書く場合，それらの違いが問題になることがあります．

たとえば，UNIXではどんなファイルでもまったく同じ形式をしていて，その管理の方法もすべての入出力に対して同じであることが原則です．しかし，他のOSではファイルの種類がいくつかあり，その呼び出し方も何種類かあるなどということがあります．よくあるのは，テキストファイルとバイナリファイルでは，読み書きの方法が違うということです．また読み書きの方法は同じでも，ファイルポインタの扱いが違うという場合もあります．この章では，どのオペレーティング・システムでも共通に扱うことができるテキストファイルの例を取り上げます．

■C言語がオープンする3つのファイル　―標準入力／標準出力／標準エラー出力―

ファイルは，必要なときに自分でオープンし，使い終わったらクローズするというのがその扱いの基本です．しかしC言語では，プログラムの起動時に自動的にオープンされ，終了時に自動的にクローズされる3つのファイルを持っています．これらは，**標準入力**，**標準出力**，**標準エラー出力**と呼ば

れます．そして**図 5-2** に示すように，標準入力はキーボードに標準出力と標準エラー出力はディスプレイに設定されています(C言語では周辺機器も1つのファイルとして扱います)．

これらのファイルは，プログラマ自身で管理する必要がありません．つまり，コンパイラが管理するので，自分でオープンしなくとも printf や scanf で画面への出力やキーボードからの入力が行えるのです．

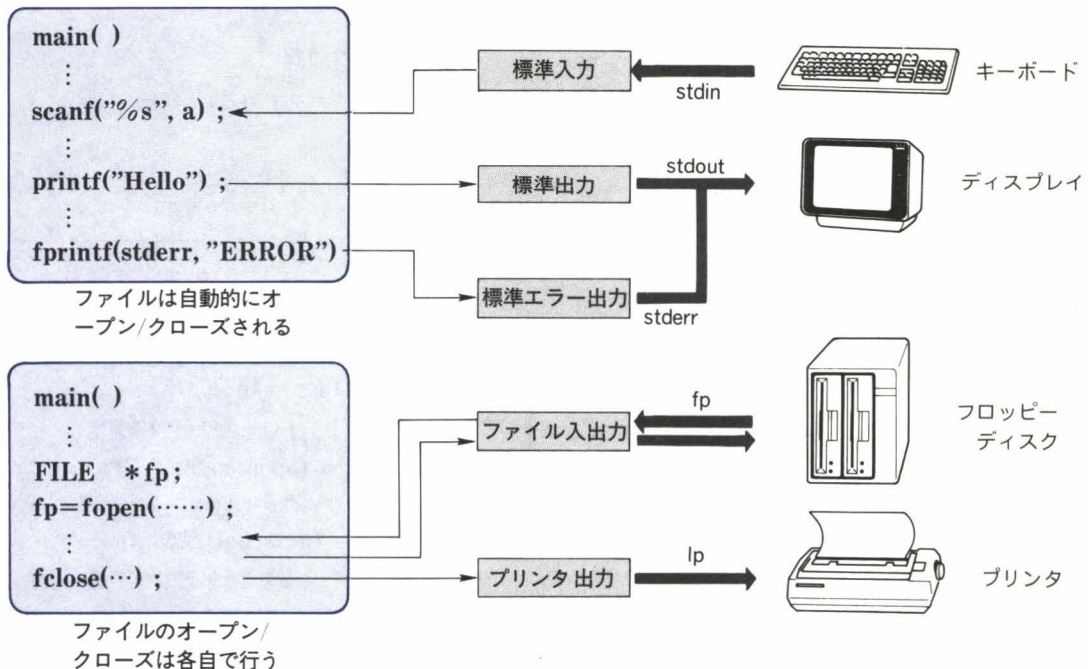



図 5-2　標準入力／標準出力／標準エラー出力

■ファイルとのデータのやりとり　―FILEストラクチャー―

実際にプログラムのなかでファイルを扱う場合には，ファイル名と読み書きするデータの位置を把握していなければなりません．C言語では，FILEという名前の**構造体**(ストラクチャ)を用いてこの管理を行います．構造体についてはここではくわしく述べませんが，〝ファイルの情報を集めたデータの集まり〟と理解してください．この構造体を使うことによって，データの集まりをまるで1つの変数の

ように扱うことができます．そして，ここでは構造体という変数(みたいなもの)のポインタのみを扱いますので，ポインタを操作するだけでファイルの操作ができます．

以下に構造体 FILE の中身とそれを使ったファイルとのデータのやりとりの例を示します(**図 5-3**)．

【構造体 FILE の中身】

1．現在のファイルを読み込み(書き出し)している位置(**ファイルポインタ**)
2．残りの文字数
3．メモリ上の１クラスタ分のバッファの位置
4．現在，読み込みをしているか書き出しをしているかの情報
5．現在のファイルが何番目にオープンされたファイルであるかという情報

この構造体全体を１つのポインタ変数で扱うことができる

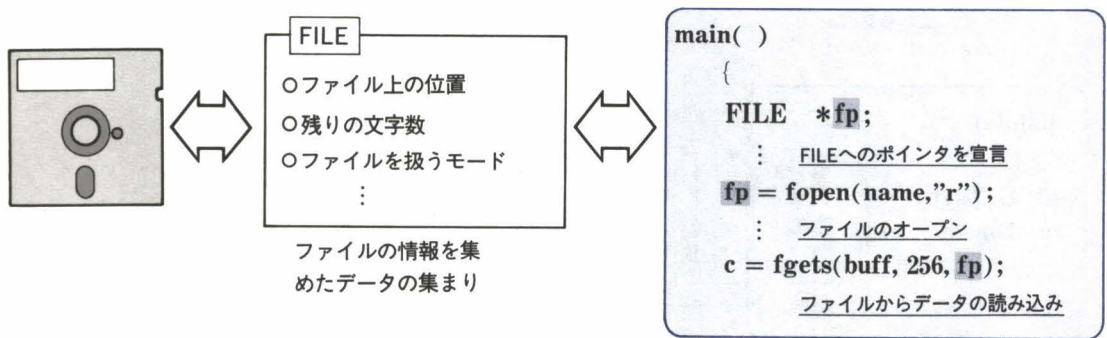



図 5-3　ファイルとのデータのやりとり

■ファイルの内容を表示するプログラム

さて，前置きが長くなってしまいましたが，ファイルの入出力を行うプログラムを組んでみましょう．ここで取り上げるのは，ファイルの内容を表示するプログラムです．MS-DOS や CP/M には，「TYPE」というコマンドがありますが，このプログラムでは１行ごとに行番号を付けて内容を表示します．

まず，プログラムを**リスト 5-1** に示します．

```
 1: /*
 2:      File-Dump Program
 3: */
 4:
 5: #include     <stdio.h> ………標準入出力関数を使用可能にする(5.3節でくわしく解説)
 6:
 7: main()
 8:      {                         ┌──大きめに宣言しておく
 9:      char              file[64]; ………処理の対象となるファイル名を入れておく文字配列
10:      unsigned int      i; ………行数を数えるカウンタ
11:      char              buff[256]; ………ファイルから読み込んだ1行の内容を入れておく文字配列(バッファ)
12:      FILE              *fp; ……構造体FILEへのポインタ
13:
14:      printf("¥n¥n"); ………2行送る
15:      printf(" Dump File-name : "); ………入力のためのメッセージを表示
16:      scanf("%s", file); ………配列fileに内容を表示するファイル名を標準入力から入力
17:      printf("¥n"); ………1行送る
18:
19:      printf("¥n DUMP-FILE : %s¥n¥n", file); ………確認のために入力されたファイル名を画面(標準出力)に表示
20:          ┌─ 5行目の"stdio.h"に定義してある
21:      if(NULL == (fp = fopen(file, "r"))) ………指定されたファイルが存在するかどうかの判断
22:           {              ………ファイルをオープンする標準関数
23:           printf("¥7¥n Cannot Open File : %s¥n¥n", file); ┐指定されたファイルが存在
24:           exit(1); ………ビープ音を鳴らすキャラクタコード         ├しない場合,メッセージを表
25:           }                                               ┘示し,プログラムを終了する
26:           ………実行を終了し,オペレーティング・システムに戻る標準関数
27:      for(i = 0 ; i < 65535 ; ++i) ………ファイルから内容を読み込んで表示するためのループ
28:           {
29:           if(NULL == fgets(buff, 255, fp)) ………ファイルの終わりかどうかの判断
30:                {
31:                break;     ………fpで指定されるファイルから,255文字もしくはキャリッジリターンまでを
32:                } ………for文を抜ける  buffに読み込む標準関数
33:           printf("%5d: %s", i+1, buff); ……… 行番号とともにbuffの内容を画面(標準出力)に表示
34:           }                  行番号  読み込んだ内容
35:      fclose(fp); ………ファイルをクローズする標準関数
36:      }
```

MSX-C, LSI C…16 行目の後に「file[strlen(file) − 1] = 0x00 ;」を追加

BDS C …………「5 : #include <bdscio.h>」に変更

「10 : unsigned i ;」に変更

「12 : FILE ibuf ;」に変更

「21 : if((−1) = = fopen(file, ibuf))」に変更

「29 : if(NULL = = fgets(buff, ibuf))」に変更

「35 : fclose(ibuf) ;」に変更

RUN/C …………5 行目を削除

「10 : unsigned i ;」に変更

「21 : if((fp = fopen(file, "r")) = = NULL)」に変更

「29 : if(fgets(buff, 255, fp) = = NULL)」に変更

リスト 5-1 ファイルの内容を表示するプログラム

プログラム名は，「catl」とします(この名前はUNIXでファイルの内容を表示するcatコマンドからとりました．行番号付きのcatという意味です)．

このプログラムの実行手順を以下に示します．

1．catl ⏎ とするとプログラムが起動する
2．「Dump File-name　:」とファイル名を聞いてくるので，ファイル名を入力する(必要ならば，ファイル名の前にドライブ名やパス名を指定する)
3．名前を入れたファイルが存在すれば，そのファイルを5桁の行番号とともに表示する．もしファイルがなければ，ビープ音を鳴らし，指定されたファイルが存在しないことを画面に表示する．

図5-4に実行結果を示します．サンプルにはこのプログラムのソースファイル自身を使っています．

```
A>catl⏎

 Dump File-name : catl.c⏎ ……… 内容を表示するファイル名を入力

 DUMP-FILE : catl.c ……………………確認のためにファイル名を表示
|←5桁→|
    1: /*
    2:      File-Dump Program
    3: */
    4:
    5: #include     <stdio.h>
    6:
    7: main()
    8:
   27:      for(i = 0 ; i < 6553
   28:           {
   29:           if(NULL == fgets(buff, 255,
   30:                {
   31:                break;
   32:                }
   33:           printf("%5d: %s", i+1, buff);
   34:           }
   35:      fclose(fp);
   36:      }

A>
```

図 5-4　リスト5-1の実行結果

さて，このプログラムのくわしい説明をしていきます．

5行目に「#include～」という実行文があります．これは，**プリプロセッサ**と呼ばれ，このプログラムで使っている標準関数を使用可能にする働きがあります．この実行文では，後に続く「stdio.h」というファイルをこの部分に取り込みます．プリプロセッサについては，この章の最後でまとめていますので参考にしてください．

12行目にFILEストラクチャを使った以下のような宣言文があります．

```
FILE      *fp;
```

ここでは，「*fp」という構造体FILEへのポインタ変数(この変数を以降，単にfpと呼びます)を使って，以後ファイル操作を行いますという宣言をしています．このポインタを使って，ファイルをオープンするとそのファイルがクローズされるまでは，この変数を用いてファイルの操作を行うことができます．

21行目からが，いよいよファイルのオープンです．

```
if(NULL == (fp = fopen(file, "r")))
```

この行では，fopenという関数を使って，これから使用するファイルのオープンを行っています．まず，この関数の仕様を以下の**表 5-13** に示します．

書　式	fopen(file, mode)	
パラメータ／返値	データ型	解　説
パラメータ1 (file)	char型のポインタ ex) char *file	オープンするファイル名の入った変数のポインタ
パラメータ2 (mode)	char型のポインタ	オープンする際のモードを指定 "r"…読み込み用(read) "w"…書き出し用(write) "a"…追加書き出し用(append)
関数の返値 (return value)	構造体FILEのポインタ ex) FILE *fp	オープンしたファイルの構造体FILEへのポインタ NULL＝ファイルが存在しない NULL≒ファイルポインタ
使用例	fp=fopen(file, "r"); fileで指定したファイルを読み込みモードでオープンする． そしてオープンしたときの構造体FILEへのポインタをfpに渡す	

表 5-13　fopen関数の仕様

ex)は，引数の型の宣言例を示します

さて，この行は内側のかっこから実行されますから，以下の２行に書き換えることができます．

```
fp = fopen(file, "r");     … 配列 file に入っているファイルを read モードでオープン
if(NULL == fp)             … fp が NULL なら以下の文を実行
```

まず，ファイル名が入っている配列のポインタとオープンする際のモード〝r〟を fopen 関数に渡します．fopen 関数はファイルをオープンできるかどうか調べ，関数の返値としてその結果を返します．ここでは，ファイルがオープンできないときには NULL が返り，オープンできるときは fp(構造体 FILE へのポインタ)が返ります．NULL は，５行目の「stdio.h」に定義してある文字列でかならず大文字で書きます(くわしくは5.3節で説明します)．ファイルが存在すると，† fp のファイルポインタ(ファイルの現在位置)は０となりファイルの先頭を指します．もしファイルが存在しなければ，fp には NULL が返るので以下の文が実行されます．

```
{
printf("¥7¥n Cannot Open FILE : %s¥n¥n", file);
exit(1);
}
```

この処理は，オープンできなかったファイル名を表示し，exit 関数を呼んでプログラムを終了させます．ここで，エラーを表示する printf 関数の文字列のなかに「¥7」という文字があります．「¥」が頭に付くと画面への制御が行われるということは２章で述べたとおりですが，この〝7〟は８進数のキャラクタコードで〝ベルを鳴らす〟という意味を持っています(209 ページ参照)．また exit 関数という標準関数は，以下のような引数をとり，プログラムを終了させます．

0 = なにもせずに抜ける

1 = エラーのリカバリをして抜ける　　注）この値はＣ言語の種類によって違う

次の for 文(27 行目)からが，ファイルの中身を読みにいくところです．

```
for(i = 0 ; i< 65535 ; ++i)
```

ここで，何行で終わるかわからないファイルに for 文で制限を設けているのは，ファイルがなんらかの原因で壊れた場合の暴走を防ぐためです．どんなファイルでも，この行数を越えることはないだろ

† fp(構造体 FILE へのポインタ)とファイルポインタを混同しないようにしてください

うという値までのループを作ります．ここでは，unsigned int 型で扱える最大値を設定しています．

通常のファイルの終わりは，for ループ中の

```
if(NULL == fgets(buff, 256, fp))  …  配列 buff に fp で示すファイルの内容を読み込む
    {                                 ファイルの終わりでは NULL が返る
    break;
    }
```

という if 文で判断し，break 文で抜けます．ここで使われている **fgets** 関数の仕様を**表 5-14** に示します．

書　式	fgets(buff, n, fp)	
パラメータ／返値	データ型	解　説
パラメータ1 (buff)	char型のポインタ ex) char ＊buff	ファイルから読み込まれたデータを保存する変数buffのポインタ
パラメータ2 (n)	int型の変数 ex) int n	ファイルから読み込むバイト数
パラメータ3 (fp)	構造体FILEのポインタ ex) FILE ＊fp	読み込むファイルのファイルポインタ
関数の返値 (return value)	char型のポインタ	NULL≒読み込みに成功 NULL＝ファイルの終わり
使用例	x＝fgets(buff, 256, fpo); fpoで指定したファイルから，256バイト(文字)までの1行のデータを読み込んで buff という変数に入れる．読み込んだ後，ファイルポインタを次の行の先頭に進める．もし，ファイルの終わりならNULLが返る	

表 5-14　fgets 関数の仕様

fgets 関数は，ファイルから指定された変数に 1 行分のデータを読み込む関数です．そして，この関数が返値として NULL を返してくるまで，1 行ごとに配列 buff にファイルの内容を読み込んでは表示するというループを繰り返します．

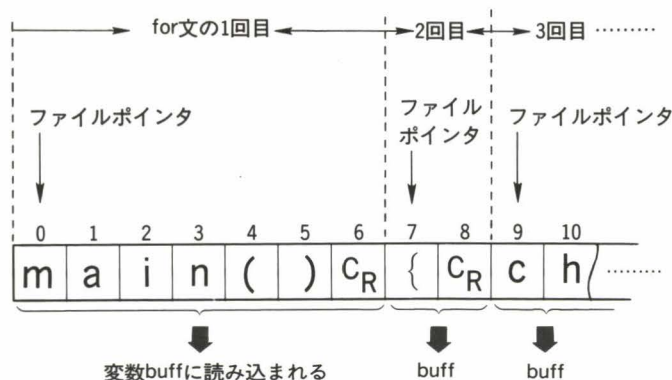



図 5-5　fgets 関数の動作

33 行目の printf 関数は，2 つの変数の内容を画面に表示します．

1 つは〝i〟という for 文のループカウンタに 1 を加えたもの(すなわち行番号)で，もう 1 つは読み込んだ配列 buff の内容そのものです．そして，この 2 つの変数を画面(標準出力)の 1 行に表示します．ここで行番号を表示する表現指示のための文字に〝5〟という数字が付いている点に注意してください．この数字は表示する桁数を指定します．つまり，

%5d　……　表示の際，値を 5 桁(右詰め)で表示する

という意味になります．

さて，ファイルが終わると，29 行目の if 文が真になり break 文が実行されます．break の意味は，「ループの制御から抜けろ」ということですから，ここでは for 文の実行単位から抜けて，35 行目の fclose 関数へと実行が移ります．

fclose(fp);　……　引数として渡されたファイルをクローズする

この下には main 関数の「}」がありますから，これでメインルーチンは終了しオペレーティング・システムへ制御が移ります．

■プログラムの改良

リスト 5-1 では，ファイルからのデータの読み込みを行いました．ここでは，ファイルへのデータの書き出しをやってみましょう．そこで，先のプログラムに手を加えて，行番号付きのリストの出力先を画面ではなくファイルになるようにプログラムを変更します．

プログラムは，**リスト 5-1** とそれほど変わりません．違っているのは，出力するための fp(構造体 FILE へのポインタ)を用意してそのファイルのオープンを行うことと，ファイルに文字列を出力する関数を用いることです．

まず，改良したプログラムと実行結果を**リスト 5-2** に示します．

```
 1: /*
 2:      File-Dump Program
 3:           Modified for FILE-output
 4: */
 5:
 6: #include     <stdio.h>
 7:
 8: main()
 9:      {
10:      char              file0[64];  } ファイル名を入れるバッファを2つ用意
11:      char              file1[64];
12:      unsigned int      i;
13:      char              buff[256];
14:      FILE              *fp0;  } 構造体FILEへのポインタを2つ用意
15:      FILE              *fp1;
16:
17:      printf("¥n¥n");
18:      printf(" Dump File-name : ");  } 行番号を付けるファイル名を入力
19:      scanf("%s", file0);
20:      printf("¥n");
21:
22:      printf(" Dest.File-name : ");  } 出力先のファイル名を入力
23:      scanf("%s", file1);
24:      printf("¥n");
25:
26:      printf("¥n DUMP-FILE : %s", file0);      } 入力されたファイル名を確認の
27:      printf("¥n DEST-FILE : %s¥n", file1);    } ため画面に表示
28:
29:      if(NULL == (fp0 = fopen(file0, "r"))) ……行番号を付けるファイルが読み込みモード
30:           {                                  でオープンできるかどうかのチェック
31:           printf("¥7¥n Cannot Open File : %s¥n¥n", file0);
32:           exit(1);
33:           }
34:
35:      if(NULL == (fp1 = fopen(file1, "w"))) ……出力先のファイルが書き出しモードで
36:           {                                  オープンできるかどうかのチェック
37:           printf("¥7¥n Cannot Open File : %s¥n¥n", file1);
38:           exit(1);
39:           }
```

```
40:
41:     for(i = 0 ; i < 65534 ; ++i)
42:         {
43:         if(NULL == fgets(buff, 255, fp0))  ………ファイルを1行ずつ読み込み,
44:             {                                  ファイルの終わりかどうかをチェック
45:             break;
46:             }                  ┌─→書き出す内容
47:         fprintf(fp1, "%5d: %s", i+1, buff);  ………出力先のファイルに書き出す
48:         }   └──→出力先の構造体FILEへのポインタ
49:     fclose(fp0); ┐
50:     fclose(fp1); ┘………ファイルのクローズ
51:     }
```

［実行結果］

```
A>catl2↵ ………ここではファイル名をcatl2とした

 Dump File-name : catl2.c↵ ………行番号を付けたいファイル名

 Dest.File-name : catl2.num↵ ………出力先のファイル名


 DUMP-FILE : catl2.c
 DEST-FILE : catl2.num

A>type catl2.num↵ ……catl2.numの内容を確認する
    1: /*
    2:      File-Dump Program
    3:           ..lified for FILE-output
    4.             fprintf(fp1,
   48:          }
   49:      fclose(fp0);
   50:      fclose(fp1);
   51:      }
```

MSX-C, LSI C…19行目の後に「file0[strlen(file0) − 1] = 0x00 ;」を追加
　　　　23行目の後に「file1[strlen(file1) − 1] = 0x00 ;」を追加
BDS C …………「6 :　#include　<bdscio . h>」に変更
　　　　「12 :　unsigned　i ;」に変更
　　　　「14 :　FILE　ibuf ;」に変更
　　　　「15 :　FILE　obuf ;」に変更
　　　　「29 :　if((−1) = = fopen(file0, ibuf))」に変更
　　　　「35 :　if((−1) = = fcreat(file1, obuf))」に変更
　　　　「43 :　if(NULL = = fgets(buff, ibuf))」に変更
　　　　「49 :　fclose(ibuf) ;」に変更
　　　　49行目の後に「putc(CPMEOF, obuf) ;」を追加
　　　　「50 :　fclose(obuf) ;」に変更
RUN/C …………6行目を削除
　　　　「12 :　unsigned　i ;」に変更
　　　　「29 :　if((fp0 = fopen(file0, "r")) = = NULL)」に変更
　　　　「35 :　if((fp1 = fopen(file1, "w")) = = NULL)」に変更
　　　　「43 :　if(fgets(buff, 255, fp0) = = NULL)」に変更

リスト 5-2　ファイルに結果を出力するプログラム

リスト 5-1 は，入力用のファイルを1つだけオープンし標準出力(画面)に結果を出力しました．このプログラムではファイルへの出力を行うので，もう1つ余分に fp とファイル名のバッファを用意します(10行目～15行目)．ここでは，それぞれの fp を「fp0」と「fp1」，ファイル名を入れるバッファを「file0[ ]」と「file1[ ]」としています．

また，ファイルをオープンする際には，29行目で file0 を読み込みモード("r")とし，35行目で file1 を書き込みモード("w")でオープンします．ここで書き込みモードでオープンすると，指定したファイルが存在した場合，以前あったファイルの内容はすべてなくなってしまうので注意してください．

47行目では，ファイルに出力するために printf に代わって **fprintf** という関数が登場しています．この関数は，fp1 で示されるファイルに printf と同じように書式指定付きの出力を行います．以下に fprintf 関数の仕様を示します．

書　式	fprintf(fp, format [, list])	
パラメータ	データ型	解　説
パラメータ 1 (fp)	構造体FILEのポインタ ex) FILE　*fp	書き出すファイルの構造体FILEへのポインタ
パラメータ 2 (format)	表現指示のための文字を含む文字列	ファイルに出力する内容
パラメータ 3 (list)	書式指定を行う引数	パラメータ2の表現指示のための文字列に対応する引数(省略可能)
使用例	fprintf(fp1, "No.%d", i); fp1で指定されるファイルに表現指示のための文字を含む文字列を書き出す	

表 5-15　fprintf関数の仕様

さて前に，「標準入力」，「標準出力」，「標準エラー出力」の3つのファイルは最初からオープンされていると述べましたが，これらを利用する場合の fp はすでに決まっています．それは，以下のとおりです．

		fpのかわりにすでに用意されているポインタ変数
標準入力(キーボード)	……………	stdin
標準出力(画面)	……………	stdout
標準エラー出力(画面)	……………	stderr

これを使って，ファイル入出力関数のファイルポインタの部分を置き換えれば標準入出力を使うことができます．たとえば31行目でエラーメッセージを表示する場合，fprintf関数を使って以下のように記述します．

```
fprintf(stderr, "¥7¥n Cannot Open File : %s¥n¥n", file0);
        ↑
        └──────── これが標準エラー出力のfp
```

■ファイルを連結するプログラム

ファイル入出力を行うプログラムをもう1つ紹介しましょう．ここで取り上げるのは，複数のファイルを連結して1つのファイルにまとめるプログラムです．

このプログラムは，「tomcat」という名前で12個までのテキストファイルを入力された順序につなげ指定したファイルに出力します．

まず始めにプログラムリストと実行結果を示しておきます(**リスト5-3**)．

```
 1: /*
 2:                 File Connection Program
 3:                            1986/02/10
 4:                            Created by N.Mita at ASCII Corp.
 5: */
 6:
 7: #include     <stdio.h>
 8:
 9: main()
10:      {
11:      FILE     *fp[12], *fpo;  ………入力ファイルと出力ファイルの構造体FILEへのポインタ
12:      char     fname0[64];     ………入力ファイルの名前が入る配列
13:      char     fname1[64];     ………出力ファイルの名前が入る配列
14:      int      i, j;           ………カウンタ用変数
15:
16:      printf("**** This is file-connection procedure**** ¥n");  }初期メッセージ
17:      printf("If end of inputs, Input-filename with '/'¥n¥n");  }
18:
19:      for(i = 0 ; i < 12 ; ++i)………ファイル名を入力し,オープンするループ(12個まで)
20:           {
21:           printf("Input file-name %3d : ", i+1); }ファイル名の入力
22:           scanf("%s", fname0);                   }
23:                                      何個目のファイル名か
24:           if('/' == fname0[0]) }
25:                break;          }………ファイル名の最初が'/'ならば入力を終えループを抜ける
26:
27:           if(NULL == (fp[i] = fopen(fname0, "r")))………オープンした結果をi番目の
28:                {                                         fpに入れる
```

```
29:             fprintf(stderr, "Cannot open file : %s¥n", fname0);
30:             i--; ………オープンできない場合は,カウンタを1つ戻す
31:             continue; ………for文の先頭(19行目)に戻る
32:             }
33:         }
34:
35:     printf("¥nInput output-file   : ");
36:     scanf("%s", fnamel);
37:
38:     if(NULL == (fpo = fopen(fnamel, "w"))) ………書き込みモードでファイルをオープン
39:         {
40:         fprintf(stderr, "¥7¥n**** Cannot open %s¥n", fnamel);
41:         exit(1);
42:         }
43:
44:     for(j = 0 ; j < i ; ++j) ………入力されたファイルの数だけ繰り返す
45:         {
46:         long    ii;
47:
48:         for(ii = 0 ; ii < 60000 ; ++ii)
49:             {
50:             int c;
51:
52:             c = fgetc(fp[j]); ……… j番目のfpで示すファイルから1文字入力
53:             if(feof(fp[j])) break; ………ファイルの終わりならば2番目のfor文を抜け62行目へ
54:             if(ferror(fp[j]))
55:                 { ………… 入力ファイルのエラーをチェックする
56:                 fprintf(stderr, "¥7¥n@@@@@@ FILE-ERROR in %d¥n", j+1);
57:                 exit(1); ………… 入力ファイルのエラーのときは,メッセージを表示してプログラムを終了
58:                 }
59:
60:             fputc(c,fpo); ………fpoで示されるファイルへ変数cの内容を書き出す
61:             }
62:         fclose(fp[j]); ………入力ファイルをクローズ
63:         printf("Copied %3d/%d¥n", j+1, i); ………コピーが終了したことを表示
64:         }
65:
66:     fclose(fpo); ………出力ファイルをクローズ
67: }
```

注記:
- 29行目: 標準エラー出力にエラーメッセージを表示
- 35〜36行目: 出力先のファイル名の入力
- 40〜41行目: オープンできない場合はメッセージを表示してプログラムを終了
- 44行目の i: ファイルをオープンしたときのループのカウンタ

[実行結果]

```
A>tomcat ↵
**** This is file-connection procedure****
If end of inputs, Input-filename with '/'

Input file-name    1 : tomcat.c↵
Input file-name    2 : catl2.c ↵     ファイル名を入力
Input file-name    3 : catl.c ↵
Input file-name    4 : / ↵ …………… 入力の終了は'/'を入力

Input output-file    : ftotal↵………出力ファイル名を入力
Copied    1/3 ………1番目のファイルのコピーを終了
Copied    2/3 ………2番目のファイルのコピーを終了
Copied    3/3 ………3番目のファイルのコピーを終了
```

```
A>type ftotal⏎ ………結果を確認してみる
/*
            File Connection Program
                          1986/02/10 Copyright CoreDump Co,.Ltd.
                          Created by N.Mita at ASCII Corp.
*/

#include    <stdio.h>

main()
    {
    FILE    *fp[12], *fpo;
    char    fname0[64];
    char    fname1[64];
    in
        fclose(fp[j]);
        printf("Copied %3d/%d¥n",         procedure**** ¥n");
        }                                  with '/'¥n¥n");

    fclose(fpo);
    }
/*
    File-Dump Program
        Modified for FILE-output
*/

#include    <stdio.h>

main()
    {
    char                fileO[64];
    char                file1[64];
    unsigned int        i;

        (fp0);
    fclose(fp1);
    }
/*
    File-Dump Program
*/

#include    <stdio.h>
              :
```

MSX-C, LSI C…22 行目の後に「fname0[strlen(fname0) −1] = 0x00 ;」を追加
36 行目の後に「fname1[strlen(fname1) −1] = 0x00 ;」を追加
「46 : unsigned int ii ;」に変更
「52 : if((c = getc(fp[j])) = = EOF ¦¦ c = = CPMEOF) break ;」に変更
「53 : putc((char)c, fpo) ;」に変更
54 行目から 60 行目を削除

BDS C …………「7 : #include <bdscio . h>」に変更
「11 : FILE fp[12], fpo ;」に変更
「27 : if((−1) = = fopen(fname0, fp[i]))」に変更

```
                「38:  if((-1) == fcreat(fname1, fp0))」に変更
                「46:  unsigned  ii;」に変更
                「52:  if((c = getc(fp[j])) == EOF || c == CPMEOF)  break;」に変更
                53行目から58行目までを削除
                「65:  putc(CPMEOF, fpo);」を追加
RUN/C ···········7行目, 46行目, 50行目を削除
                「14:  int  i, j, c;     long  ii;」に変更
                「27:  if((fp[i] = fopen(fname0, "r")) == NULL)」に変更
                「38:  if((fpo = fopen(fname1, "w")) == NULL)」に変更
DeSmet C ·········「52:  if(0x1a == (c = fgetc(fp[j])))  break;」に変更
                53行目から58行目までを削除
```

リスト 5-3 tomcat プログラムと実行結果

プログラムの解説をしていきましょう.

11行目では，このプログラムで扱う構造体 FILE へのポインタ変数を用意しています.

```
FILE   *fp[12], *fpo;
```

入力用のファイルとして「*fp」を12個(ここでは配列で扱います)と，出力用のファイルとして「*fpo」を宣言します.

19行目以降は，ファイル名を変数に取り込みオープンできるかどうかチェックして，fp にその値を代入しているループです. 24行目の if 文では，入力ファイル名の終わりを表す〝/″を判断しています. また, 29行目では前述の fprintf 関数を使って，ファイルをオープンできなかった場合のメッセージを標準エラー出力に出しています.

31行目では, continue 文が使われています. break 文が実行単位の外に制御を移動するのに対して，この文は，この文が含まれる最も内側の for 文や while 文の先頭に処理を移します.

continue ;　……　この文が含まれる最も内側の for 文または while 文に処理を移す

つまり，この文が実行されると19行目に制御が移ることになります. ループの途中で先頭に戻りたいときにはこの continue 文を使います.

44行目からが，入力されたファイルを出力先にコピーするループです. 44行目の変数 i は19行目でファイル名の入力を行ったときのカウンタですから，入力されたファイルの数だけループを繰り返すことになります. 48行目からの2番目の for ループが，実際にファイルから1文字読み込んでは出力ファイルに1文字ずつ書き出す処理を行います. ここで1文字入出力には, fgetc と fputc の2つの標準関数を使っていますので，その仕様を挙げます(**表 5-16, 表 5-17**).

この関数を使う場合，50行目にあるように1文字入力を受けとる変数cをchar型ではなくint型で宣言する点に注意してください(**表5-16**の関数の返値を参照)．

また，ファイルの終わりと入力ファイルのエラーをチェックするためにfeofとferrorという標準関数を53行目と54行目に使っています．これらの仕様も以下に示しておきます(**表5-18，表5-19**)．

書　式	fgetc(fp)	
パラメータ／返値	データ型	解　説
パラメータ1 (fp)	構造体FILEのポインタ ex) FILE ＊fp;	読み込むファイルの構造体FILEへのポインタ
関数の返値 (return value)	int型の変数 ex) int c;	読み込んだ文字
使用例	c = fgetc(fp); fpで指定したファイルから1文字読み込み，変数cに返値として読み込んだ文字を返す．また読み込んだ後，ファイルポインタを1つ進める．ファイルの終了やエラーには，"feof"，"ferror"関数を使う	

表5-16　fgetc関数の仕様

書　式	fputc(c, fp)	
パラメータ／関数	データ型	解　説
パラメータ1 (c)	int型の変数 ex) int c;	ファイルに書き出す文字
パラメータ2 (fp)	構造体FILEのポインタ ex) FILE ＊fp;	書き出すファイルの構造体FILEへのポインタ
関数の返値 (return value)	int型の変数	書き出した文字列
使用例	fputc(c, fp); fpで指定したファイルへ変数cで指定した1文字を書き出す．書き出した後ファイルポインタを1つ進める．ファイルの書き込みエラーの判定には，"ferror"関数を使う	

表5-17　fputc関数の仕様

書　式	feof(fp)	
パラメータ／関数	データ型	解　説
パラメータ1 (fp)	構造体FILEのポインタ ex) FILE ＊fp	ファイルの終わりのチェックを行う 構造体FILEへのポインタ
関数の返値 (return value)	int型の変数	0 ＝ファイルの終わりではない 0 ≒ファイルの終わり
使用例	while(!feof(fp))……… fpで指定されたファイルが終わりでない間，以下の処理を行う	

表 5-18　feof関数の仕様

書　式	ferror(fp)	
パラメータ／型	データ型	解　説
パラメータ1 (fp)	構造体FILEのポインタ ex) FILE ＊fp	ファイルのエラーのチェックを行う 構造体FILEへのポインタ
関数の返値 (return value)	int型の変数	0＝ ファイルにエラーなし 0≒ ファイルにエラーあり
使用例	if(ferror(fp))……… fpで指定されたファイルにエラーがあった場合，以下の処理を行う	

表 5-19　ferror関数の仕様

■高水準入出力関数と低水準入出力関数

この節で紹介した入出力関数は，すべて**高水準入出力関数**と呼ばれています．これに対して**低水準入出力関数**も存在しますが，めったに使われません．低水準入出力関数は，最も基礎的な入出力関数群で，マシンのハードウェアに大きく依存しており，さまざまな制限があります．C言語によっては，このレベルで他のC言語との互換性がとれないものも多いので，通常のプログラミングでは，ここで紹介した高水準入出力関数を使うようにしてください．

5.3 プリプロセッサ

プリプロセッサ(pre-processor)とは,「前処理」のことでコンパイラがプログラムをコンパイルする前に行う処理を意味します．このプリプロセッサを使うと，指定したファイルの取り込みや文字列の置き換えを行うことができます．これはC言語で最も便利な機能の1つです．ここでは，プリプロセッサの使い方を学んでいくことにしましょう．

■ プリプロセッサの書式

前節で取り上げたプログラムの先頭にはおまじないのように，

という行が含まれていました．この「#」で始まる行がプリプロセッサで，コンパイラに前処理を行うように指示します．

C言語のプログラムは基本的にフリーフォーマットですが，このプリプロセッサだけは「行」を単位としています．つまり，

```
#include
        <stdio.h>
```

と書いてはいけません．また，コンパイラによっては「#」を1カラム目に書かなければならないものもあるので注意してください．

■ プリプロセッサの種類とその使い方

プリプロセッサには多くの種類がありますが，ここではC言語のプログラムでよく使われる代表的なプリプロセッサを3つ紹介します．通常のプログラミングでは，この3つを知っていればまず不便なことはありません．

◆#include ― 指定されたファイルの取り込み

#includeは，この後に続く「＜」と「＞」の間に指定したファイルをプログラムリストのこの部分へ読み込んでくる指示を与えます(**図 5-6**).

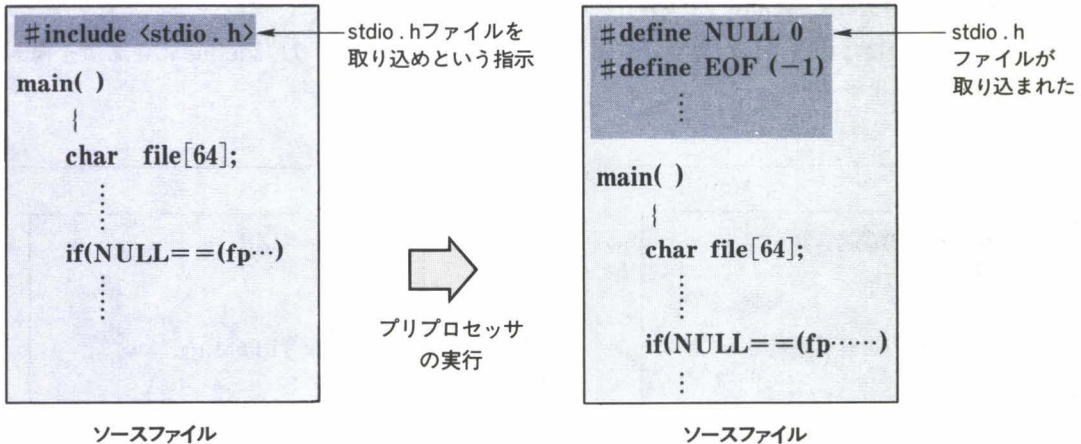



図 5-6 ＃include によるファイルの取り込み

表 5-20 に，#includeの書式と使用例を示します．取り込むファイル名の指定には，「＜ ＞」を使う場合と「" "」を使う場合があります．UNIXやMS-DOSのように階層ディレクトリを持つオペレーティング・システムの場合，この2つの方法はファイルをどこから捜すのかが異なりますので注意してください(階層ディレクトリを持たないOSでは関係ありません).

＃include ファイル名	指定されたファイルを取り込む
使用例	＃include 〈stdio.h〉 stdio.hというファイルを取り込む ＃include "ioctl.h" ioctl.hというファイルを取り込む
解　説	〈ファイル名〉…あらかじめ指定されたディレクトリからファイルを捜す "ファイル名"…今いるディレクトリからファイルを捜す 〈注意〉階層ディレクトリを持たないオペレーティング・システムではこの区別はない

表 5-20 ＃include の書式と使用例

#includeで取り込まれるファイルのうち「.h」の拡張子の付いたものをC言語では**ヘッダファイル**と呼びます(「.h」が付いていなくともよいのですが，一般的な約束としてそうなっています)．このヘッダファイルについては，後でくわしく解説します．

◆#define　— 文字列の置き換え(マクロ定義)

#defineは，この後に指定された文字列の置き換えを指示します(**図5-7**)．#defineによる置き換えを**マクロ定義**と呼びます．

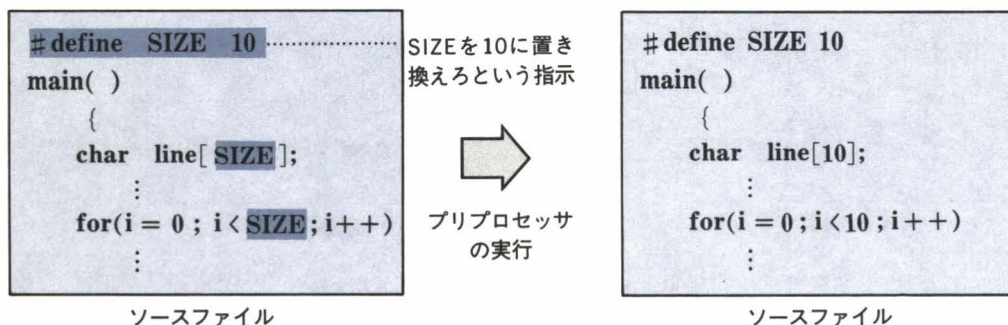



図 5-7　#defineによる文字列の置き換え

#defineで〝置き換わる文字列〟は，大文字で書きます．これはプログラムを見たときに，この文字列が別に指定してある文字列と置き換わることを明確にするためのC言語の約束です．

表5-21に#defineの書式と使用例を示します．

#define　文字列1　文字列2	文字列1を文字列2に置き換える
使用例	#define　BS　0x08 　　BSを0x08に置き換える #define　SIZE　100 　　SIZEを100に置き換える
解　説	文字列の置き換えに使用する．文字列1は，通常大文字で記述する

表 5-21　#defineの書式と使用例

#define をよく使うのは，プログラムで使っている定数の置き換えです．たとえば，キャラクタコードをわかりやすい文字に置き換えてプログラムを読みやすくすることができます．また，何度も出てくる定数を #define で定義しておくと，後でその大きさを変更するときにこの #define 文を変えるだけで済みます．

◆#if，#else，#endif ―条件付きコンパイル

このプリプロセッサは，プログラム中の3カ所に置かれ，コンパイラにコンパイルする範囲を指示します．これは，デバッグ時やコンパイラの種類によってプログラムの書き方を変えなければならないときによく用いられます．以下にその書式と使用例を示します(**表 5-22**)．なお，#else は省略することもできます．

#if 条件… #else… #endif	条件が真ならば#if～#elseまでをコンパイルし，偽ならば#else～#endifまでをコンパイルする． #elseは省略可能
使用例	#define DEBUG 1 /* デバッグ時は1とする */ : #if DEBUG printf("¥n value A is : %d ¥n",a); } デバッグ時にコンパイルされる範囲 : #else : b = a; : #endif :
解　説	このプリプロセッサは，デバッグ時，コンピュータの違いやコンパイラの違いなどを吸収するときに使われる

表 5-22 #if，#else，#endif の書式と使用例

■ヘッダファイルの中身

#include 文で取り込まるファイルは，ヘッダファイルと呼ばれます．ここでは，その中身をのぞいて見ることにしましょう．

この章の標準関数を使ったプログラムでは，#includeを使って[†]「stdio．h」というファイルを取り込みました．多くのC言語コンパイラではこのファイルが用意されています．この「stdio」とは，「Standard Input Output」の略で**標準入出力**ということです．つまり〝入出力に関係する値や関数の定義を行うファイル〃です．ほとんどのシステム標準関数を使用する場合には，このヘッダファイルを取り込む必要があります．

このファイルは，**図5-8**のようになっています．

```
/*
 * stdio.h
 *
 * defines the structure used by the level 2 I/O ("standard I/O") routines
 * and some of the associated values and macros.
 *
 * (C)Copyright Microsoft Corporation 1984, 1985
 */

#define  BUFSIZ  512
#define  _NFILE  20
#define  FILE    struct _iobuf
#define  EOF     (-1)

#ifdef M_I86LM
#define  NULL    0L
#else
#define  NULL    0  ………NULLが定義されている
#endif

extern FILE {  …………………構造体FILEの定義
        char *_ptr;
        int   _cnt;
        char *_base;
        char  _flag;
        char  _file;
        } _iob[_NFILE];

#define  stdin   (&_iob[0])  ┐
#define  stdout  (&_iob[1])  ├標準入出力の構造体FILEへのポインタ
#define  stderr  (&_iob[2])  ┘
#define  stdaux  (&_iob[3])
#define  stdprn  (&_iob[4])
```

```
         _IONBF      0x04
#define  _IOMYBUF    0x08
#define  _IOEOF      0x10
#define  _IOERR      0x20
#define  _IOSTRG     0x40
#define  _IORW       0x80
```

```
#define  getc(f)     (--(f)->_cnt >= 0 ? 0xff & *(f)->_ptr++ : _filbuf(f))
#define  putc(c,f)   (--(f)->_cnt >= 0 ? 0xff & (*(f)->_ptr++ = (c)) : ¥
                     _flsbuf((c),(f)))
#define  getchar()   getc(stdin)
#define  putch   ( )  putc((c),stdout)
         fileno(f)   ((f)
#d                                    IOEOF)
   function declarations for those who   p)
 * on arguments to library function calls
 */

#ifdef LINT_ARGS                 /* arg. checking enabled */

void clearerr(FILE *);
int fclose(FILE *);
int fcloseall(void);
FILE *fdopen(int, char *);
int fflush(FILE *);
int fgetc(FILE *);
int fgetchar(void);
char *fgets(char *, int, FILE *);
int flushall(void);
FILE *fopen(char *, char *);
int fprintf(FILE *, char *, );
int fputc(int, FILE *);
int fputchar(int);
int fputs(char *, FILE *);
int fread(char *, int, int, FILE *);
FILE *freopen(char *, char *, FILE *);
int fscanf(FILE *, char *, );
int fseek(FILE *, long, int);
long ftell(FILE *);
int fw                  FILE *);
cha   n FILE *fopen(), *freopen
  ttern long ftell();
extern char *gets(), *fgets();

#endif  /* LINT_ARGS */
```

各種の標準関数が定義されている

図 5-8 stdio.hの中身(Microsoft C Compilerの場合)

ここでは，#defineでよく使われる定数の置き換えが行われています．前述のプログラムで使った「NULL」や構造体「FILE」は，ここに定義されていたのです．

このように，ヘッダファイルは#define文の集まりですから，自分でよく使うマクロ定義などを集めて自分だけのヘッダファイルにしておくと同じようにして一度にプログラム中に取り込めるので便利です．

† BDS Cでは，「stdio.h」の代わりに「bdscio.h」を用いる．またRUN/Cでは，標準関数を使用する場合も「#include <stdio.h>」を行う必要はない

【この章のポイント】

1．ファイルの入出力には，構造体 FILE へのポインタを使用する

2．ファイルの読み書きには，標準入出力関数を使う

3．C 言語には，プログラム実行時に自動的にオープンし終了時にクローズする 3 つのファイルがある

標準入力 ……　**キーボード**　　**標準エラー出力** ……　**ディスプレイ**

標準出力 ……　**ディスプレイ**

4．ヘッダファイルの取り込みや文字列の置き換えには，「#include」，「#define」プリプロセッサを使う

［**練習問題**］

1．あるファイルをもう 1 つの別のファイルへコピーするプログラムを，できるだけ短く書きなさい

2．1 で作ったプログラムをもとにして任意の文字をコピー中に消すプログラムを作りなさい（必要な値はキーボードから入力する）

注：

1 の例題をやるには，2 の例題のことを考えて作ってみましょう．また，プログラムを 1 つ作成するごとに，後々でも使える「自分の関数」が 1 つできるように(つまり自分の財産を増やすことを考えて)プログラムを作成しましょう．

（解答は219ページ）

第6章

C言語による
プログラム開発の実際

これまでの章で，C言語のプログラムを記述するための最低限必要な要素の説明は終わりました．この章では，作成したプログラムをコンパイルして実行するまでの手順をもう一度振り返ってみたいと思います．

これまでサンプルプログラムを打ち込んでは，コンパイルという作業を何度も繰り返してきた人は，なぜこんなにめんどうな作業が毎回いるのだろうと思ったことでしょう．C言語はコンパイラ型の言語ですから，コンパイルという作業は避けて通れないのですが，実はそれによって生まれるメリットは実行速度の向上だけではないのです．

それは，プログラムのリンク(結合)が行えることです．これまでは，1つのソースファイルから1つの実行形式のファイルを作るだけなので，そのメリットは活かしていません．しかし，自分(または他人)が作った関数やFORTRANなど他のコンパイラ型の言語で書いたプログラムをリンクすることによって，1つの実行形式のファイルを作ることができます．そうすれば関数を再利用したり，C言語で記述しにくい部分を他の言語で書くといったことが行えます．

作成したプログラムを実行するまでに，もう1つ忘れてはならないことは，プログラムのエラーを取ることです(このことをよくデバッグといったりします)．コンパイルを実行してみるとわけのわからないエラーがいっぱい出てしまったというのは，誰でも一度は経験します．多くのC言語ではエラーメッセージが不親切なので，ちょっとしたエラーを取るのもかなり苦労します．そしてコンパイルを何度も繰り返しては，またエラーの山を築くことになります．

エラーを取るにも，コツがあります．それさえ覚えておけばエラーメッセージも恐れるに足りません．ここでは，そんなノウハウもまとめてみました．

また，最後にエラーを事前にチェックするためのプログラムと，文字列を検索するプログラムを紹介しておきます．

6.1 コンパイルとリンク

これまでは，実行形式のファイルを作成するためにバッチファイルやコンパイル用のコマンドを用いてきました．ここでは，そこでやっている手順を1つ1つ追いながら，実行形式のファイルがどのように作られるかを見てみましょう．

■実行形式のファイルを作成する手順

C言語で私たちが書くプログラムは，**ソースファイル**(Source file)と呼ばれ，人間が書けて読める形になっています．当然のことながら，このファイルを直接コンピュータは実行できません．そこで，コンパイルという作業を行って，実行形式のファイルを作成します．ここでは例として，Microsoft C Compilerでバッチファイルを用いてコンパイルする場合を考えてみます(**図6-1**)．　ここでの解説は，コンパイル用のコマンドを用いた場合でも基本的に同じですから，お手持ちのC言語で実際に確認をしながら読んでみてください．

図 6-1　コンパイルの実行

このバッチファイルでは，2つのことが行われています．1つは**コンパイル**という作業です．もう1つは，**リンク**という作業です．ここでは，なんだかよくわからないオプションが後ろに付いています．

つまり，ソースファイルから実行形式のファイルを作るためには，この2つの作業を行う必要があるのです．なぜこんなめんどうなことになっているのかを以下で解説していきます．

■コンパイル　―ソースファイルからオブジェクトファイルを作るまで―

多くのコンパイラでは，コンパイルという作業のみではコンピュータが実行できるファイルは作れません．では，いったい何をするのか，コンパイルという作業だけを実行させてみます(**図6-2**)．

```
A>msc b:test.c, b:test.obj;⏎ ………コンパイルだけを実行してみる
Microsoft C Compiler Version 3.00
(C)Copyright Microsoft Corp 1984 1985

A>dir b:⏎

  ドライブ B: のボリュームラベルは MSC
  ディレクトリは B:¥

TEST     C        35  86-02-09   3:33 ………ソースファイル
TEST     OBJ     293  86-02-09   4:26 ………オブジェクトファイルが作成されている
        2 個のファイルがあります
 1247232 バイトが使用可能です
A>
```

図 6-2　コンパイルの実行

ここで作られるファイルを**オブジェクトファイル**(正式には**リロケータブル・オブジェクト・モジュール**<relocatable object modules>)と呼びます．このオブジェクトファイルは，実行形式のファイルとなる一歩手前のいわば中間ファイル的な存在です．

このファイルは実行形式のファイルと違って，メモリ上でプログラムの置かれる位置(アドレス)が決まっていません．実行形式のプログラムは，かならず"どのアドレスで動くか"という指定がされているのですが，このファイルはそれがまだ決まっていないのです．つまり，どのアドレスにも置くことができるので，リロケータブル(再配置可能)と呼ばれます．

このコンパイルまでの手順を**図6-3**に示しておきます．

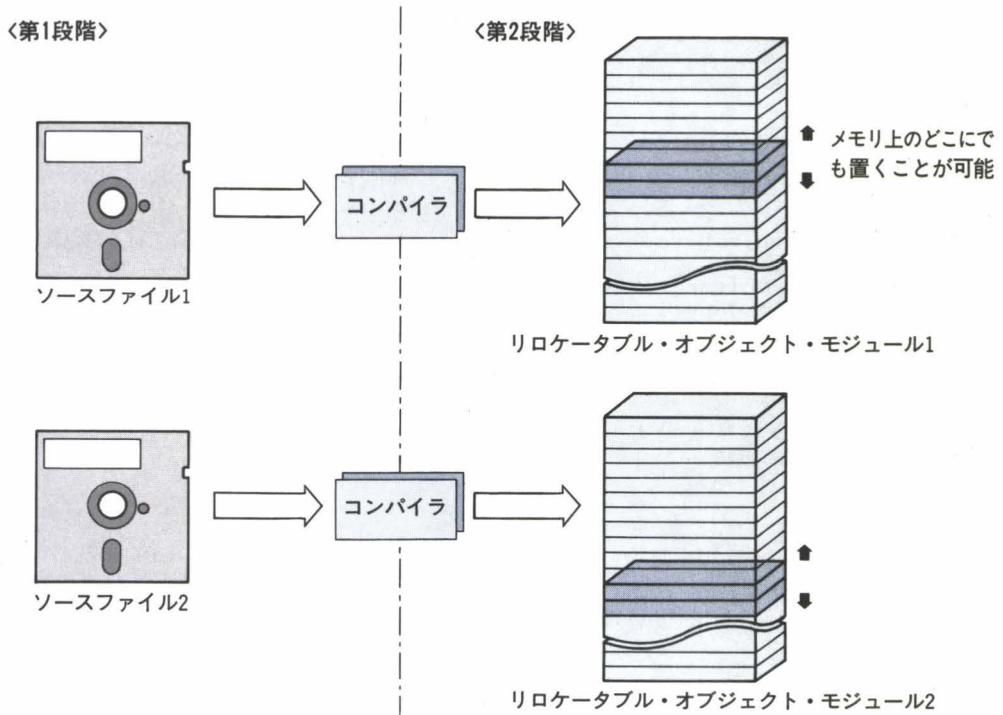



図 6-3　プログラム開発の流れ(1)

しかし，なぜこのような〝いらないファイル〟が必要なのでしょうか．まず，この段階で保存されたプログラムは管理が簡単になります．これらのファイルは，実行ファイルよりもサイズが小さくなりますから，保存に場所をとりません．一度作ってしまった関数で改良の必要がないものはこの形で取っておくことができます．たとえば，C言語の標準関数の多くは，オブジェクトファイルの集まりとして供給されています．そして，後で必要なときに次に述べるリンクという作業を行うことで，1つの実行プログラムとしてまとめることができるのです．また，機械語に近い形をしていますから，他の言語のコンパイラやアセンブラで作ったプログラムといっしょにすることも可能です．

これが(C言語に限らず)，オブジェクトファイルの存在意義となります．

■リンク　－オブジェクトファイルから実行形式のファイルを作るまで－

さて**図 6-2** では，オブジェクトファイルを作るところまでを実行しました．その続きをやって，実行形式のファイルを作ってみましょう(**図 6-4**)．

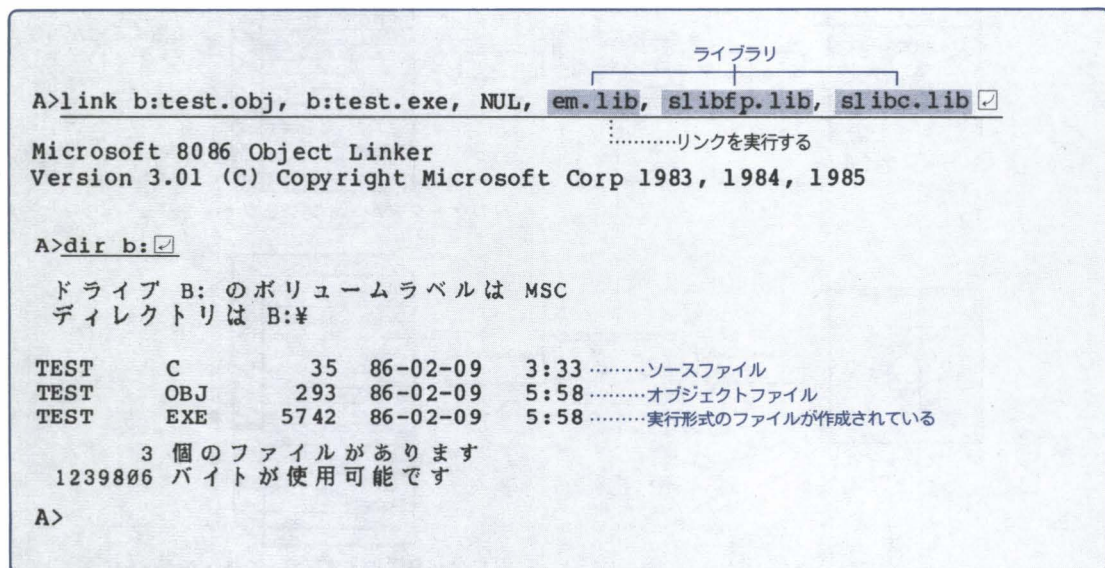



図 6-4　リンクの実行

ここで行われる作業は，**リンク**(link)と呼ばれます．また，リンクを実行するプログラムを**リンカ**(linker)と言います．リンクとは，〝つなげる〟という意味ですから，さきほどの再配置可能なオブジェクトファイルをつなぎ合わせて，実行形式のファイルを作り上げることになります．ここでは，プログラムがメモリ上のどこに置かれるかというアドレスも決定されます．これによって，はじめて動くプログラムができあがることになります．

また前述したように，コンパイルが終わったオブジェクトファイルは，C言語で作ったものであろうとFORTRANで書かれたものであろうと同じ形式をしていますから，ここでお互いにつなげてしまうこともできます．ただし，どんな場合でも可能というわけではなく制限はあります．

リンクを行って実行形式のファイルができるまでの手順を**図 6-5** に示します．

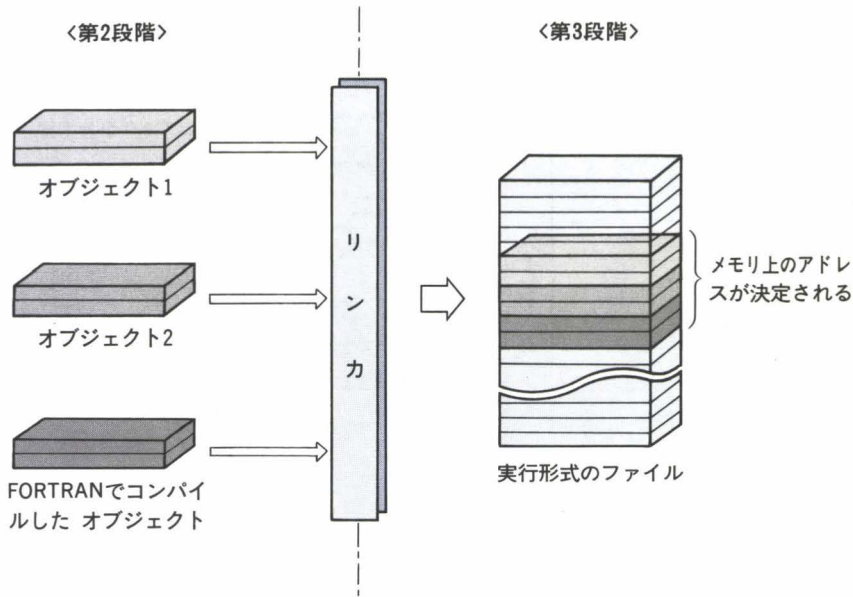



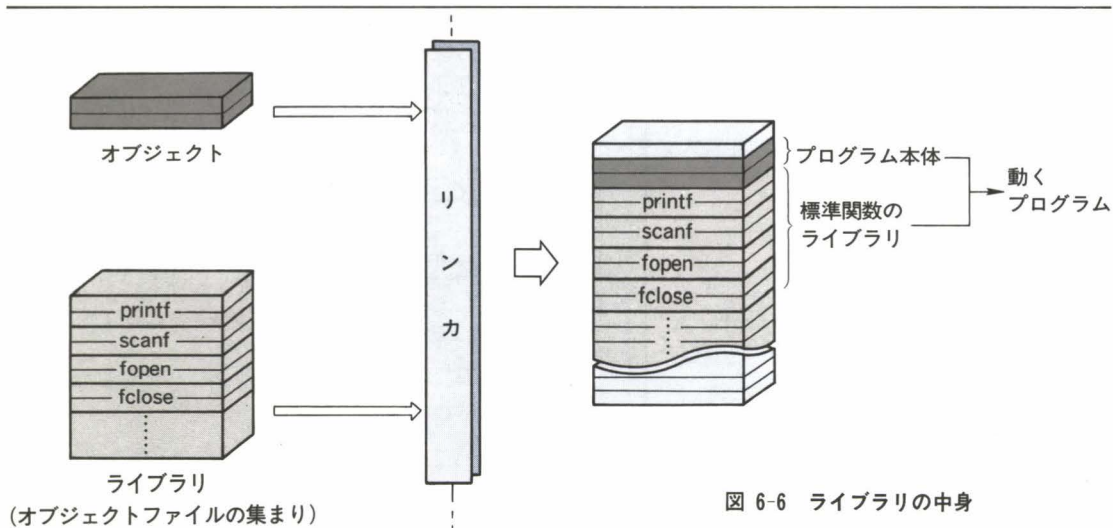
図 6-5 プログラム開発の流れ(2)

■ライブラリ —標準関数のオブジェクトファイルの集まり—

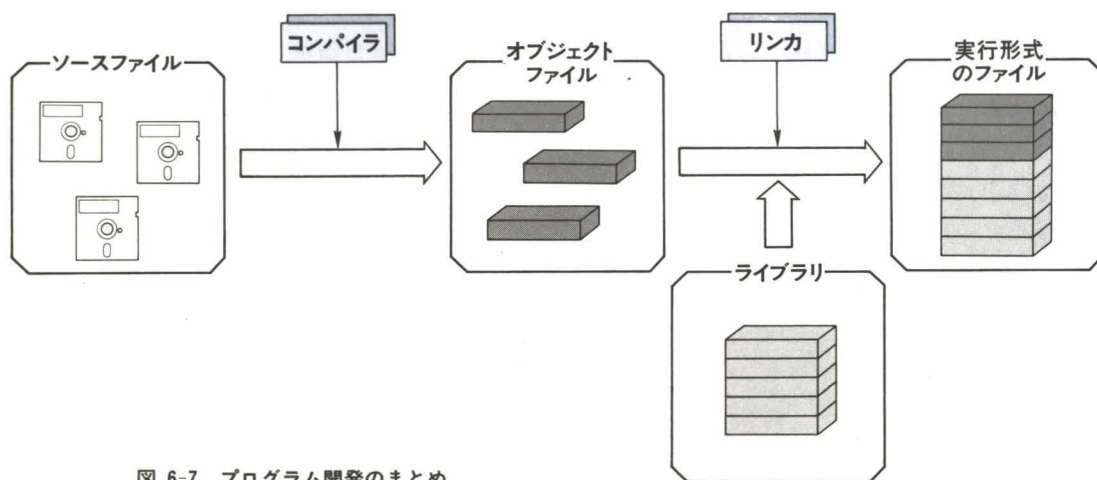
先の**図 6-4** では指定したオブジェクトファイルは１つですが，実は別のオブジェクトファイルとつなげられて実行形式のファイルができています(**図 6-6**).

このファイルは，「～.LIB」という名前でリンクを実行するときに指定するものです．これを**ライブラリ** (library) と呼びます．ライブラリは，C言語コンパイラのメーカーが供給するオブジェクトファイルの集まりであり，先に説明したオブジェクトファイルと形式的には同じものです．そのなかには，「printf」などの標準関数をコンパイルしたものがたくさん詰まっています．標準関数を使って作ったプログラムは，このライブラリといっしょになって最終的な〝動くプログラム〟となるわけです．

ライブラリとしては，通常はメーカーが供給したものを使いますが，自分で作ったオブジェクトファイルを集めてライブラリを作ることもできます．C言語によっては，ライブラリを作成するツールであるライブラリ・マネージャというプログラムが付属しているものもあります．

図 6-6　ライブラリの中身

これまで見てきたように，実行形式のファイルを作るまでには，かなり複雑な手順を踏んでいることがわかったと思います．本書では1つのソースファイルしか取り上げませんが，多くのソースファイルから1つの実行形式のファイルを作ることを**分割コンパイル**と呼びます．この分割コンパイルをスムーズに行えることがC言語(コンパイラ言語)の特徴でもあります．

最後にもう一度，C言語のプログラム開発の手順をまとめておきます(**図 6-7**)．

図 6-7　プログラム開発のまとめ

6.2　エラーの対処法

プログラムにはエラー(間違い)がつきものです．〝絶対に大丈夫〟だと思って，いざ実行してみると山のようにエラーが出てしまったというのは誰でも経験します．

ここで大事なことは，〝エラーの出ないプログラムを書くこと〟ではなく，〝エラーを見つけやすいプログラムを書くこと〟です．いつ，どこで，誰が見ても何をしているかが，一目瞭然であるプログラムを書くことが，エラーを少なくする最も有効な対策なのです．

ここでは，C言語のエラーメッセージの見方とその対処法について解説していきます．

■エラーメッセージの見方

エラーを取るためには，コンパイラがどのようなエラーのときどんなメッセージを出すのかを知っていなければなりません．しかし，一口に〝エラーメッセージ〟といってもC言語によって，かなりの違いがあります．たとえば，BASICなどでもおなじみの，

```
SYNTAX  ERROR
```

から，そのままではさっぱりわからないものまで，さまざまなメッセージが現れます．

C言語のコンパイラが出すエラーメッセージは，大きく分けて以下の2つの種類があります．

〈エラーメッセージの種類〉

1. **ワーニング(Warning)**
   警告．たいしたことではないが，コンパイラとしては気になって仕方がないので，一応警告しておくという意味
2. **エラー(Error)**
   間違い．コンパイラとしては，これ以上作業を続けられないといってサジを投げてしまったという意味

よく，〝エラーが出た！〟といってはあわてて心当たりの箇所を手直し，コンパイルをやり直して，またエラーを出すというパターンを繰り返す人がいます．このほとんどが，コンパイル時に出てくるメッセージをあまり読まずに手直しを始めることが原因です．まずメッセージが出てきたら，それは「Warning」なのか「Error」なのかの判断ぐらいはつけておくことです．

メッセージが出た場合には，以下のように考えて対処すればよいでしょう．

〈エラーメッセージの見方〉

1. ワーニング

すべてのメッセージがワーニングならば実行形式のファイルができているので，とりあえず実行しきちんと動くかどうかをチェックする．正しく動作しない場合は，ワーニングの出た箇所に誤りがある可能性が高いのでその行を調べる

2. エラー

メッセージのなかにエラーが含まれる場合は実行形式のファイルが作成されないので，エラーの出た箇所を調べてプログラムを修正し，コンパイルをやり直す

ついでですが，〝日本人がコンピュータを学ぶ場合は，英語をめんどくさがってはいけない〟という鉄則があります．わからない単語は，辞書で引けばよいのです．何も英語が得意である必要はありません．めんどうでも辞書を引くことです．

■デバッグの考え方

BASICなどのインタープリタは，行単位に実行されエラーが起きた時点で止まるようになっています．ところがコンパイラ型の言語では，エラーのあるなしにかかわらず最後までコンパイルを行ってから一度にメッセージを表示します．また，エラーがあるうちは，とりあえず実行してみるというわけにはいきません．

エラーを取ることを**デバッグ**(虫取り)と呼んでいますが，C言語でのデバッグにはちょっとしたコツがあります．このコツを知らないと，何度もコンパイルを繰り返して無駄な時間を費やしてしまうことになります．

それでは具体的に，エラーメッセージの事例を見ていくことにしましょう(**リスト6-1**)．ここではMicrosoft C Compilerを用いた例を示しますが，エラーの対処法自体はどのコンパイラでも共通ですので各自で確かめてみてください．

```
1: /*
2:     File-Dump Program
3: */
4:
5: #include    <stdio.h>
6:
```

```
 7: main()
 8:     {
 9:     char            file[64];
10:     unsigned int    i ←………セミコロンが抜けている
11:     char            buff[256];
12:     FILE            *fp;
13:
14:     printf("¥n¥n");
15:     printf(" Dump File-name : ");
16:     scanf("%s", file);
17:     printf("¥n");                ……ダブルクォーテーションが抜けている
18:
19:     printf("¥n DUMP-FILE : %s¥n¥n, file);
20:
21:     if(NULL == (fp = fopen(file, "r")))
22:         {
23:         printf("¥7¥n Cannot Open File : %s¥n¥n", file);
24:         exit(1);
25:         }
26:
27:     for(i = 0 ; i < 65534 ; ++i)
28:         {
29:         if(NULL == fgets(buff, 255, fp))
30:             {
31:             break;
32:             }
33:         printf("%5d: %s", i+1, buff);
34:         }
35:     fclose(fp);
36:     }
```

[実行結果]

```
A>cc catl.c⏎

A>msc b:catl.c, b:catl.obj;
Microsoft C Compiler Version 3.00
(C)Copyright Microsoft Corp 1984 1985
b:catl.c(20) : warning : newline in string constant ………ワーニング(警告)
b:catl.c(11) : type `char' unexpected
b:catl.c(21) : syntax error: `if'
b:catl.c(21) : syntax error: `)'
b:catl.c(27) : syntax error: `for'
b:catl.c(27) : expected '(' to follow `i'        …………エラー(間違い)
b:catl.c(27) : syntax error: `rel op'
b:catl.c(27) : syntax error: `incr op'
b:catl.c(32) : syntax error: `}'
b:catl.c(34) : syntax error: `}'
b:catl.c(36) : syntax error: `}'
                                   └─エラーの内容
                    └─エラーの種類
A>        └─エラーの発見された行
```

リスト 6-1 エラーメッセージの事例(Microsoft C Compilerの場合)

リスト 6-1 は，前章の〝ファイルの内容を見るプログラム〞に2カ所手を加えて，エラーメッセージの例としたものです．たった2カ所のはずのエラーが，これだけのメッセージとなって現れてきました．しかし，このほとんどは〝エラーが生んだエラー〞であり，本当のエラー以外は，すべて〝エラーの分家，息子，孫〞なのです．そしてエラーの元を絶てば，すべてが消えてしまうものばかりです．

このままでは，同じような顔をしたエラーのどれが本命なのかさっぱりわかりません．しかし，このようなエラーには，決まった対処法があります．

多くのC言語の処理系では，エラーは**リスト 6-1** のように行番号付きで表示されます．そこで，BASICと同じように行単位でエラーを見ていくことになります．以下にエラーの対処法を示します．

【C言語コンパイラのエラー対処法】

1. とりあえず，Warningは無視する

2. エラーの表示された行番号のうち最も若い行番号に着眼して次の点を見る

- ・かっこ，とくに「{」と「}」がきちんと対応しているか
- ・カンマ「,」，セミコロン「;」，ダブルクォーテション「"」が抜けていないか，または不適当ではないか
- ・変数名や予約語(while，forなど)のスペルミスがないかなど

3. 2に間違いがなければ，その行番号の1つ前の行を見て2と同じ点を調べる

最初に気をつけなければならないのは，以上の点です．これをフローチャートにしてみます(**図 6-8**)．

さて，これを**リスト 6-1** の例に当てはめてみましょう．まず最初のメッセージは，「Warning」なので無視するとして，次のメッセージに注目します．該当する行(11行目)を見ると誤りがないので，その1つ前の行を見ます．すると最後の「;」が抜けていることに気づきます．次のメッセージは21行目ですがこれは2つ前の行(20行目は空行)を調べると誤っていることがわかります．このようにエラーメッセージを見ていくと誤りは以上の2箇所で，それを修正しコンパイルをやり直すとメッセージがすべて消えて実行形式のファイルが作られます．

C言語は〝構造化言語〞なので，その構造の区切りとなるかっこやセミコロンなどの記号は，重要な意味を持っています．プログラムを1つの構築物とすると，これらの記号はその骨組みをつなぐリベットのようなものです．1つ抜けると，小さな部品でも構築物全体がガタガタになってしまいます．

C言語の最初のエラーの多くは，このような区切り記号を忘れたというエラーに尽きるようです．

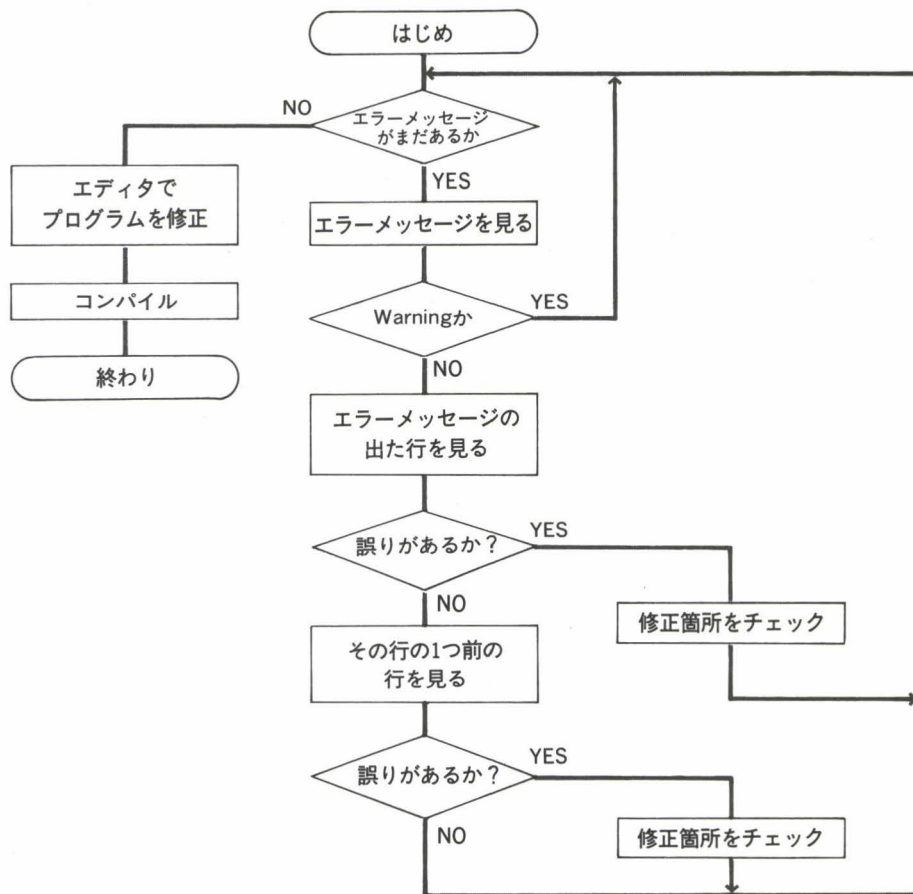



図 6-8 エラーの対処

その他のエラーは，他の言語でもよく起きるものです．たとえば，

・予約語(while，for など)の使い方の誤り
・数値の型が宣言と一致しない
・関数の引数の型が呼ばれる側と一致しない

などがあります．前述の対処法でも取れないエラーは，この点にも気をつけて見てみましょう．

■リンク時のエラー

実行形式のファイルを作るためには，コンパイルのほかにリンクという作業が必要です．コンパイルは成功しても，リンク時にエラーが起きてしまうことがあります．たとえば，以下の**リスト 6-2** のような事例を見てみましょう．

```
 1: /*
 2:         File-Dump Program
 3: */
 4:
 5: #include    <stdio.h>
 6:
 7: main()
 8:         {
 9:         char            file[64];
10:         unsigned int    i;
11:         char            buff[256];
12:         FILE            *fp;
13:
14:         printf("¥n¥n");
15:         printf(" Dump File-name : ");
16:         scanf("%s", file);
17:         pirntf("¥n");
18:           └────スペルミスをしてしまった
19:         printf("¥n DUMP-FILE : %s¥n¥n", file);
20:
21:         if(NULL == (fp = fopen(file, "r")))
22:             {
23:             printf("¥7¥n Cannot Open File : %s¥n¥n", file);
24:             exit(1);
25:             }
26:
27:         for(i = 0 ; i < 65534 ; ++i)
28:             {
29:             if(NULL == fgets(buff, 255, fp))
30:                 {
31:                 break;
32:                 }
33:             printf("%5d: %s", i+1, buff);
34:             }
35:         close(fp);
36:         }
```

［実行結果］

```
A>cc b:catl↵

A>msc b:catl.c, b:catl.obj;
Microsoft C Compiler Version 3.00      } コンパイルは成功する
(C)Copyright Microsoft Corp 1984 1985  }
```

```
A>link b:catl.obj, b:catl.exe, NUL, em.lib, slibfp.lib, slibc.lib

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

Unresolved externals: ………………………オブジェクトファイルのなかでリンクできない関数が発見された
  ┌→スペルが誤っている
_pirntf in file(s): ………………………リンクできない関数の名前
 B:CATL.OBJ(b:catl.c)

There was 1 error detected

A>
```

リスト 6-2 リンク時のエラー(Microsoft C Compilerの場合)

リスト 6-2 は前述の**リスト 6-1** と同じプログラムですが，今度は関数名を間違えてしまいました．この場合は，コンパイル時にエラーが発見できず実行形式のファイルを作るリンク時にエラーが現れます．コンパイル時にエラーが出ないのは，最終的な実行形式のファイルができるまで，関数はその実体がわからなくてもよいからです．前節のコンパイルとリンクの話を思い出してください．printf などの標準関数や自分で作った関数は，リンク時にまとめられて1つの実行形式のファイルとなります．コンパイル時には関数のチェックを行わないのでスペルミスをしてもそのような関数があると判断されます．ところが，実際にリンクする時になってその関数がどこにも存在しないとエラーとなりメッセージが表示されるのです．そしてリンク時のエラーは，この関数名のスペルミスという場合がほとんどです．

リスト 6-2 の実行結果を見てみると，リンクできない関数がメッセージとなって現れています．コンパイル時と違って，ソースファイルの行番号は出てきませんが，エディタでその文字列を検索し修正してもう一度コンパイルを行えば実行形式のファイルができます．

■実行時のエラー

エラーのなかで一番困るのは，〝エラーメッセージの出ないエラー〞です．つまり，コンパイルとリンクは成功して実行形式のファイルはできるが，いざ実行してみるとエラーが起きてしまう場合です．このようなエラーは，へたをすると暴走という事態になり，リセットボタンを押してもう一度起動し直すはめになります．

C言語では，とくに以下のようなエラーはメッセージとなって現れませんので注意が必要です．

1. **数値のオーバーフロー**
   - ・「int」の大きさの数を扱うのに，「char」で宣言した
   - ・「unsigned」で宣言しなければいけないのに「signed」で宣言したなど
2. **ポインタとして宣言した数を，ポインタでない数として扱った**
3. **宣言した配列の大きさを越える領域を使ってしまい，他の変数を壊してしまった**
4. **コメントの始まりの「/＊」を書いて，締めくくりの「＊/」を忘れている**

以上は文法上の誤りですが，そのほかアルゴリズム(プログラムの論理的構造)の欠陥で間違った結果が出てしまうこともあります．その場合には，BASICなどと同じようにプログラム中にprintf関数を入れて変数の値を表示させてみるなどの対処法が必要になってきます．

以上が主要なエラーとその対策ですが，C言語の場合，〝こんなメッセージが出たらこうする〞という細かな対処法はあまり功を奏しません．それよりも，多くのプログラムを書いてみることです．そうすれば，エラーを取るコツが自然とつかめてきます．

6.3　エラーをチェックするプログラム

C言語は，かっこが構造の区切りとなり，多くのエラーもこのかっこに起因することは前の節で述べたとおりです．C言語で書かれたソースファイルは，かならずかっこの対応が付いていなければなりません．つまり，〝左向きと右向きのかっこの数が一致しないプログラムはエラーを含んでいる〟ということです．

そこで，ここではデバッグのときに便利な「かっこチェッカ」プログラムを作ってみます．

■ プログラムの仕様

プログラムを書くにあたって，まず「仕様書」を書いておきましょう．仕様書といってもとくに細かく規定する必要はありません．以下のことを注意しておけばよいでしょう．

1．機能
2．使用法
3．外部への影響

まず最初の機能は，指定されたソースファイルを読み込んで，左右のかっこ(〝{ }〟，〝[ ]〟，〝( )〟)を数えて表示し，同じ種類のかっこの数が合わないときにメッセージを表示します．ただし，コメント中のかっこは数に入れません．次の使用法は，プログラム名を「kakko」として，チェックするファイル名はプログラムから聞いてくるものとします．また，このプログラムでは余計なファイルを作らず，オペレーティング・システムに影響を及ぼすこともないので外部への影響はありません．

■ プログラムのアルゴリズム

はじめに，プログラムのアルゴリズムについて簡単に説明しておきます．このプログラムは，ファイルから1文字ずつ読み込んで，かっこかどうかの判断をしてその数を数えています．

このプログラムの特徴は，コメントの部分のかっこを数えないところです．最初のうちは，その処理を理解するのがちょっとむずかしいかもしれません．以下に，読み込んだ文字がコメント中かどうかを判断するアルゴリズムをフローチャートで示してみます(**図 6-9**)．

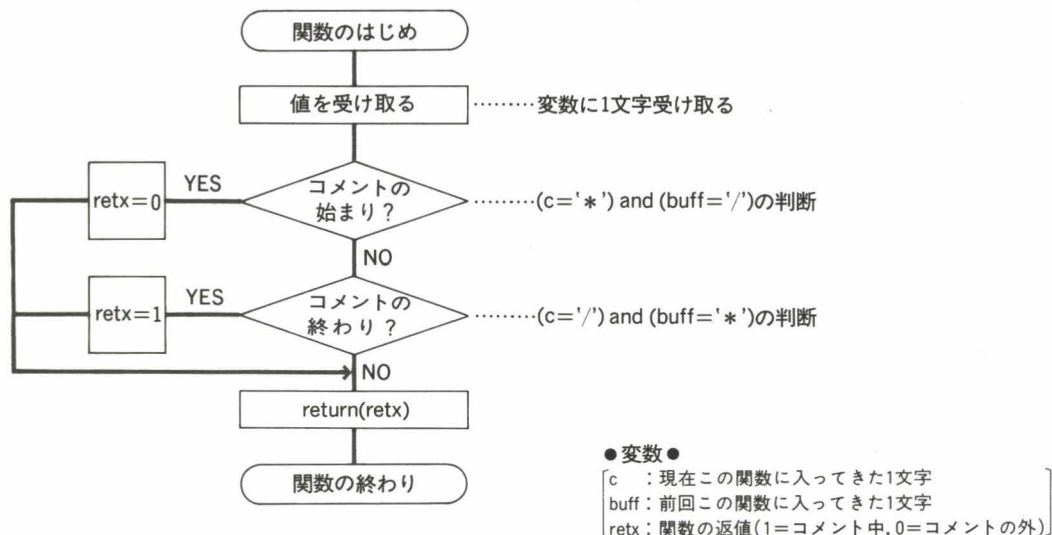



図 6-9　コメント中かどうかを判断するアルゴリズム

コメントのなかにいるかどうかを判断するには，コメントの始め(/＊)と終わり(＊/)の文字を認識しなくてはなりません．この文字が1文字であれば処理は簡単なのですが，ここでは2文字を判断しなければならないので，前の1文字を取っておく余分なバッファを用意する必要があります．

この部分は関数として記述しておくと，後でほかのプログラムでも同様な処理を行いたい場合に使うことができます．

以下に「kakko」プログラム全体のフローチャート(**図 6-10**)と，プログラムリストおよび実行結果(**リスト 6-3**)を示します．

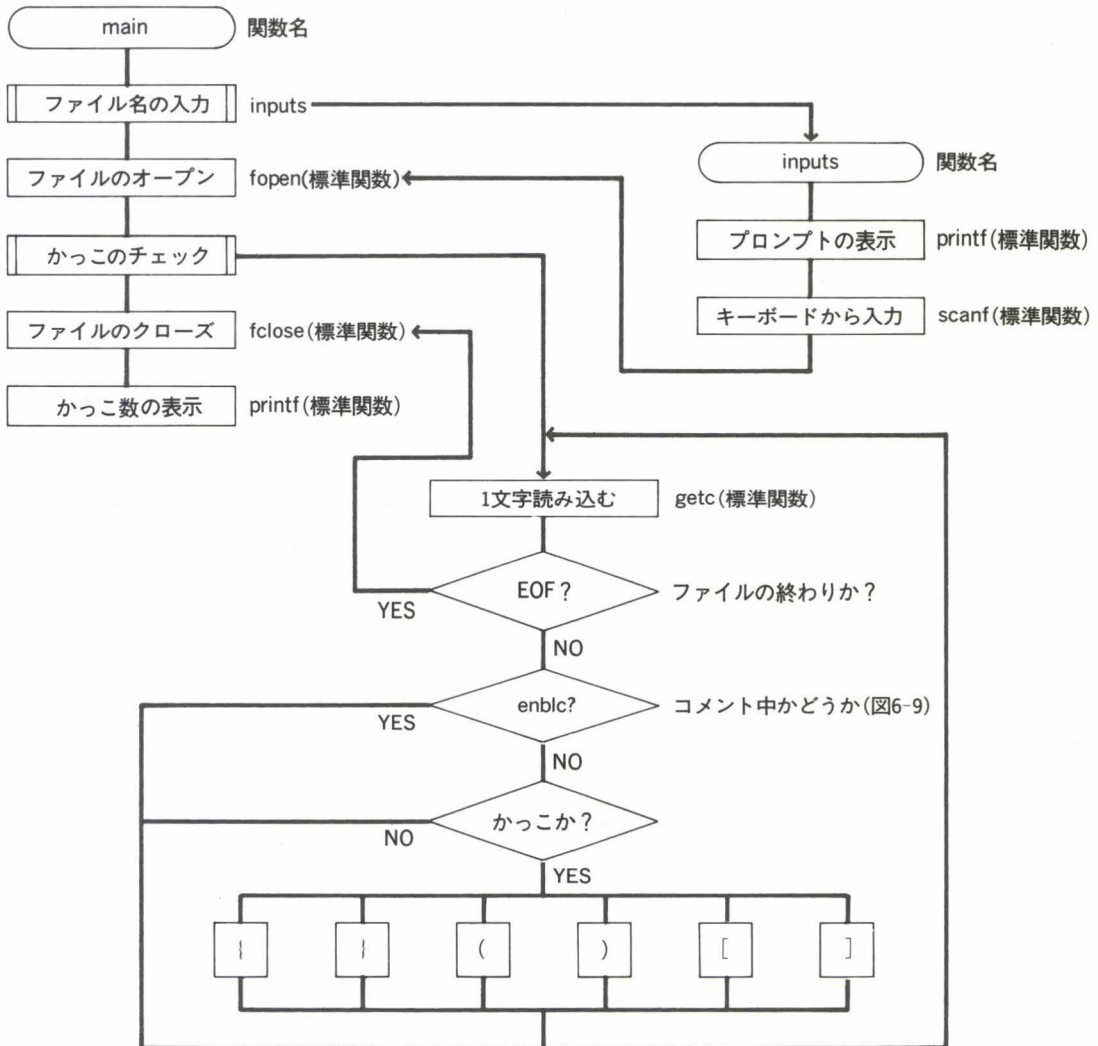



図 6-10 kakkoのフローチャート

```
 1: /*
 2:                   kakko checker program
 3:                                     created by N.Mita     1985/12/31
 4:
 5:     This program check the number of { ,},[,],( and ).
 6:     This is useful programs to syntax-checks of C-lang.
 7: */
 8:
 9: #include         <stdio.h> ………標準関数を使うために必要なstdio.hファイルをこの部分に取り込む
10:
11: main()
                                        {  }  (  )  [  ]
12:     {                               |  |  |  |  |  |
13:     static  int    kakko[6] = { 0,0,0,0,0,0 }; ………各かっこの数を数える
14:                                                 カウンタを0で初期化
15:     long     i; ………ファイルの文字数を数えるカウンタ
16:     FILE     *fp; ………構造体FILEへのポインタ変数を宣言
17:     char     file[64]; ………入力されるファイル名が入るバッファ
18:
19:     inputs("¥n **** check file name : ", file); ………inputs関数を使って
20:                                                       ファイル名を入力
21:     if(NULL == (fp = fopen(file, "r"))) ………ファイルを読み込みモードでオープンできるか
22:         {                                      どうか調べる
23:         printf("¥n¥7***** Cannot OPEN file : %s¥n", file); } オープンできない
24:         exit(1);                                             } 場合の処理
25:         }                    ループカウンタのiを符号付き整数として宣言したので,
26:                              正負の数が明確になる16進数で表記した
27:     for(i = 0 ; i < 0x0FFFFFF0 ; ++i) ………ファイルを1文字ずつ読み込むループ
28:         {
29:         int    c; ………読み込んだ文字を入れておくバッファ
30:                 ………stdio.hに定義してある文字列
31:         if(EOF == (c = getc(fp))) break; ………1文字ずつ読み込み,ファイルの終わりならば
32:                           ……1文字入力用標準関数     for文を抜ける
33:         if( 1 == enblc(c))
34:             {    ………コメント中かどうかを調べる(0:コメント中, 1:コメントの外)
35:             switch(c)
36:                 {          配列変数にもインクリメント演算子を使うことができる
37:                 case '{': kakko[0] ++; break;
38:                 case '}': kakko[1] ++; break;
39:                 case '(': kakko[2] ++; break;    読み込んだ文字が, かっこであれば
40:                 case ')': kakko[3] ++; break;    該当する配列のカウンタを1つ進める
41:                 case '[': kakko[4] ++; break;
42:                 case ']': kakko[5] ++; break;
43:                 default : break;
44:                 }                       switch文を抜ける
45:             }
46:         }
47:
48:     fclose(fp); ………ファイルをクローズする
49:
50:     printf(" { : %4d    %4d : }", kakko[0], kakko[1]); ……左右のかっこの数を表示
51:     if(kakko[0] != kakko[1])
52:         printf("¥7 *** unmatched { and }.¥n"); } 数が異なる場合はビープ
53:     else                                       } 音を鳴らして警告を表示
54:         printf("¥n");
```

```
55:
56:     printf(" ( : %4d    %4d : )", kakko[2], kakko[3]);
57:     if(kakko[2] != kakko[3])
58:             printf("¥7 *** unmatched ( and ).¥n");
59:         else
60:             printf("¥n");
61:
62:     printf(" [ : %4d    %4d : ]", kakko[4], kakko[5]);
63:     if(kakko[4] != kakko[5])
64:         printf("¥7 *** unmatched [ and ].¥n");
65:     else
66:         printf("¥n");
67:     }
68:
69: inputs(line,ans) ……………標準入力(キーボード)から,文字変数の配列に値を入力する関数
70:     char    line[]; ………表示するコメントを配列として受け取る
71:     char    ans[]; …………入力された文字列を配列で返す
72:     {
73:     printf("%s",line); ……コメントの表示
74:     scanf("%s",ans); ………キーボードから入力された文字の取り込み
75:     printf("¥n");
76:     }
77:
78: enblc(c) ………コメント中かどうかを判断する関数
79:     int     c; ……… 入力された1文字を受け取る
80:     {
81:     static  int     retx = 1; ………コメント中かどうかを示すフラグ. 最初は1で初期化しておく
82:     static  char    buff = '0'; ……1つ前の文字が入るバッファ. 最初は'0'を入れておく
83:                     (1:コメントの外
84:     switch(c)       (0:コメントの中
85:         {  …… この判断はif文でも記述できるが,switch~case文を使った方がわかりやすい
86:         case '*': if(buff == '/') retx = 0; break;……"/*"なのでコメントの始まり
87:         case '/': if(buff == '*') retx = 1; break;……"*/"なのでコメントの終わり
88:         default : break;     ………1つ前の文字
89:         }                    (86-87: 現在の文字)
90:     buff = c; ………次の処理に備えて,現在の文字をバッファに保存する
91:     return(retx); ………関数の返値として変数retxを返す
92:     }
```

[実行結果]

```
A>kakko⏎

**** check file name : kakko.c⏎ ………kakko.cのかっこをチェックしてみる

{ :   12      12 : }
( :   39      39 : )
[ :   25      25 : ]
```

(25 / 25 ← プログラム中で使用されている回数)

MSX-C, LSI C…「15: unsigned int i;」に変更
74行目の後に「ans[strlen(ans)-1] = 0x00;」に変更

BDS C …………「9: #include <bdscio.h>に変更
「10: int retx; char buff;」を追加

```
                「13:  int  kakko[6];」に変更
                「15:  unsigned  i;」に変更
                「16:  FILE  ibuf;」に変更
                18行目の前に「retx = 1;     buff = '0';」を追加
                「18:  for(i = 0;i<6;++i)  kakko[i] = 0;」を追加
                「21:  if((-1) == fopen(file, ibuf))」に変更
                「31:  if((c = getc(ibuf)) == EOF ¦¦ c == CPMEOF)  break;」に変更
                「48:  fclose(ibuf);」を追加
                81行目，82行目を削除
RUN/C ………… 9行目，29行目を削除
                「17:  char  file[64], c;」に変更
                「21:  if((fp = fopen(file, "r")) == NULL)」に変更
                「27:  for(i = 0;i<0x0FFFFFF0l;++i;)」に変更
```

リスト 6-3　kakkoのプログラムリストと実行結果

■ プログラムの解説

このプログラムの重要な点について，いくつか解説を加えておきます．細かな点については，プログラム中に示してあります．

まず，13行目に4章で使ったstatic変数が出てきます．ここでは，各かっこの数を入れるint型の配列kakkoを"0"で**初期化**しています．static変数は，auto変数と違って共有スペースに変数の領域が取られるのではなく，各変数ごとに1つの領域が確保されるので，このようにあらかじめ値を設定しておくことができます．以下に，static変数の初期化についてまとめておきます．

【static型変数の初期化】

1．文字の場合

```
static  char  buff = 'a';   … 1文字の場合はシングルクォーテーションを使う
```

2．文字列の場合

```
static  char  buff[20] = "This is buffer.";  …文字列の終わりには
                                                かならず"0"(ヌル文字)が入る
```

3．数値の場合

```
static  int  buff = 12;
```

4．数値配列の場合

```
static  int  buff[5] = {1,2,3,4,5};  … カンマで区切って"{ }"で囲む
```

注)配列以外の場合も，"{ }"で囲んで初期化を明示することがある

```
ex)  static  char  buff[20] = {"This is buffer."};
```

また81行目～82行目のenblc関数のなかでも，コメント中かどうかを表す関数の返値retxと1つ前に読み込まれた文字を取っておく変数buffをstatic変数として宣言しています． static変数は，さきにも述べたように1つの変数に1つの領域が確保されますから，この場合のように関数を抜けた後も値をとっておきたいときに使うことができます．

19行目では，文字列入力用の関数として作った「inputs」を使って，ファイル名を変数に取り込んでいます．それ以降のファイルをオープンするときの処理はこれまでと同じです．

31行目でオープンしたファイルから文字を読み込むのに，標準関数である「getc」を使っています．この関数は，1文字入力用の関数です．**表6-1**に，その仕様を示します．

書　式	getc(fp)	
パラメータ／返値	データ型	解　説
パラメータ1 (fp)	構造体FILEへのポインタ ex) FILE *fp	読み込むファイルの構造体FILEへのポインタ
関数の返値 (return value)	int型の実値	−1＝ファイルの終わり −1≒読み込み成功
使用例	c = getc(fp); fpで指定したファイルから，1文字読み込んで，変数cに入れる．もし，ファイルの終わりならcには−1が入る	

表6-1 getc関数の仕様

また，その行に出てくる〝EOF〟は〝End Of File〟を意味し，ファイルの終わりを表すマクロとして「stdio.h」ファイルに定義されています．

35行目のswitch～case文では，かっこの種類に応じてカウントを行っています．ここで「kakko[0]++;」というように配列の要素に対して，インクリメント演算子を使っている点に注意してください．また，86行目～87行目では，case文のなかにif文を使って1つ前の文字を判断しています．

6.4 文字列を検索するプログラム

大きなテキストファイルを眺めるときに，特定の文字列を検索できると便利です．また，C言語のソースファイルでも，ある変数や予約語が使われている箇所を調べる際に使うことができます．

この節では，テキストファイルから文字列を検索するプログラムを紹介します．このような汎用的なユーティリティ・プログラムは，いくつか用意しておくと後で非常に役に立ちます．

■ プログラムの仕様

このプログラムは，テキストファイル中の文字を検索して文字列が含まれる行を行番号を付けて表示するものです．文字列を表示する際には，検索文字がその行の何カラム目にあるのかを次の行で"^"の記号を付けて示します．また，これまでと同様に検索の対象となるファイルや文字列は，プログラムの実行時に聞いてくるものとします．

プログラム名は，「cgrep」としましょう．この名前は，UNIXのユーティリティにある文字列検索コマンドである「grep」にちなんだものです．

■ プログラムのアルゴリズム

プログラムリストを示す前に，プログラムの重要なポイントをまとめておきます．

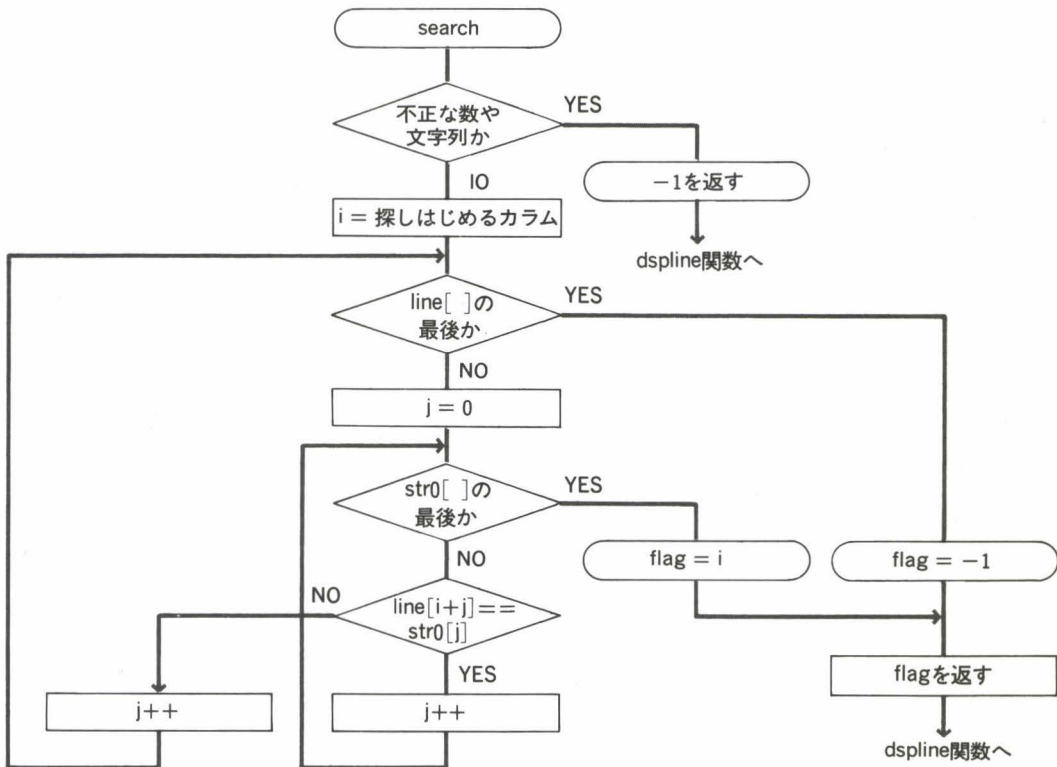
このプログラムは，3つの関数から構成されています．1つは，読み込んだ行のなかから指定された文字列を捜すsearch関数です．この関数では，文字列を捜すだけでなく見つかった文字列の先頭のカラム数を返すようにします．search関数のフローチャートを以下に示します(**図 6-11**)．

・line[ ]……検索対象文字列
・str0[ ]……検索文字列
・flag…………文字列が一致したか
　　　　　　どうかを表すフラグ
　（−1—一致しない
　　n—一致したときのカラム

図 6-11　search関数のフローチャート

もう1つの関数は，画面表示用のdspline関数です．この関数では，まず検索文字列の含まれる行を表示し，次の行に〝^〟記号で検索文字の先頭に印を付けて表示します．dspline関数のフローチャートは以下のようになっています(**図 6-12**)．

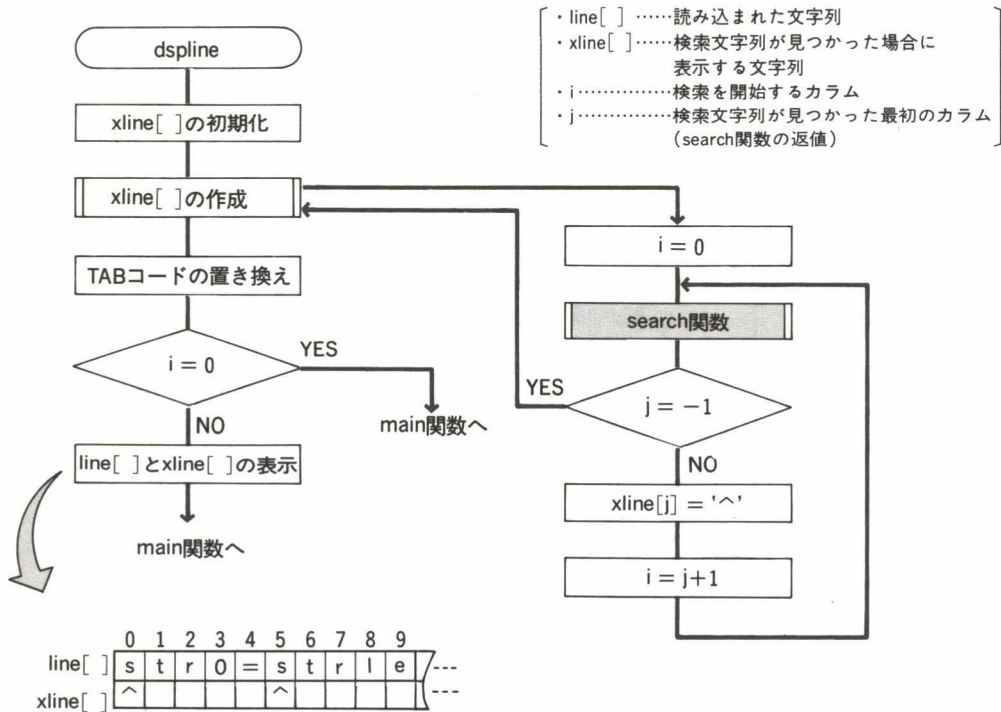



図 6-12　dspline関数のフローチャート

最後の関数は，前述の2つの関数を呼び出して制御するmain関数になります．また，ここではファイル名と検索文字列をキーボードから取り込み，ファイルのオープンも行います．

この3つの関数を使った「cgerp」全体のフローチャートを**図 6-13**に示します．

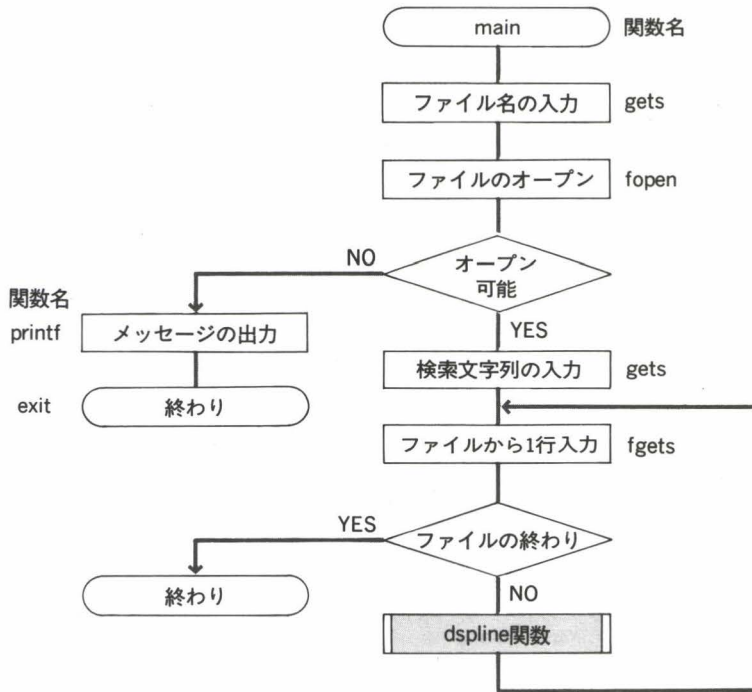



図 6-13 cgrepのフローチャート

cgerp プログラムは，これまでのプログラムと比べると少し大きくなっています．search 関数のアルゴリズムがむずかしいかもしれませんが，残りはゆっくり眺めればわかるはずです．以下にcgrep プログラムとその実行結果を示します．

```
 1: /*
 2:        String search procdeure for TEXT-FILE
 3:                        Created by N.Mita as ASCII-Corp.
 4:                                            1986/02/02
 5: */
 6:
 7:
 8: #include    <stdio.h>………標準入出力関数を使うためにに必要なファイルをこの部分に取り込む
 9:
10: #define HTAB     0x09………HTABを0x09(タブコード)に置き換える
11:
```

```
12: main()
13:     {
14:     char     fname[64]; ……………ファイル名を入れるバッファ
15:     char     str_in[64]; ……………検索文字列を入れるバッファ
16:     char     line_in[256];………ファイルから読み込んだ文字列を入れるバッファ
17:     int      x; ……………………………読み込んだ行数を数えるカウンタ
18:     FILE     *fp; ………………………構造体FILEへのポインタ変数
19:
20:
21:     printf("¥n***** Input File-name            : ");
22:     gets(fname);
23:     if(NULL == (fp = fopen(fname, "r")))
24:         {
25:         printf("¥n¥7****** No such-file : %s¥n", fname);
26:         exit(1);
27:         }
28:
29:     printf("***** Input strings for search   : ");
30:     gets(str_in);
31:     printf("¥n");
32:
33:     for(x = 0 ; x < 32000 ; ++x)
34:         {
35:         if(NULL == fgets(line_in, 255, fp)) break;………fpで示すファイル
36:         dspline(line_in, str_in, x);                   から1行(255文字まで)
37:         }                                              をline_inに読み込む
38:
39:     fclose(fp);
40:     }
41:
42: dspline(line, str0, lnum) ………検索結果を1行ずつ画面に表示する関数
43:     char     line[];
44:     char     str0[];
45:     int      lnum; ………行数は,変数そのものが渡される
46:     {
47:     int      i, j; ………内部で使うカウンタ用の変数を用意
48:     char     xline[256]; ………検索結果を表示するための文字配列を用意
49:
50:     for(i = 0 ; i < strlen(line) ; ++i)
51:         xline[i] = ' ';
52:
53:     xline[strlen(line)] = 0;………文字列の最後には,"0" を代入しておく(Cストリング)
54:     i = 0;………カラム数を数えるカウンタを0にセット
55:
56:     while((-1) != (j = search(i, line, str0)))………文字列の終わりまで検索を繰り返す
57:         {
58:         xline[j] = '^';
59:         i = j + 1;
60:         }
61:
```

21～27行の注釈：ファイル名を入力し，ファイルをオープンする

29～31行の注釈：検索文字列を入力する

36行の注釈：読み込んだ文字列と検索文字列，そして行番号を持ってdspline関数を呼ぶ

43～44行の注釈：読み込んだ文字列と検索文字列は，ともに配列のポインタが渡されてくる

50行 strlen(line) の注釈：lineの文字列の長さ

50～51行の注釈：xline[ ]は読み込まれた文字列と同じ長さの空白がつまった配列として初期化する

56行 search(i, line, str0) の注釈：カラム数を示すカウンタと読み込んだ文字列，検索文字列をsearch関数に渡し何カラム目で一致するかどうか調べる

58行の注釈：見つかった場合，対応するxline[ ]の配列に印を付ける

59行の注釈：見つかった箇所から1つカラムを進める

```
 62:     for(j = 0 ; j < strlen(line) ; ++j)……………文字列をはじめから
 63:         if(HTAB == line[j]) xline[j] = HTAB;     検索しタブコードを探す
 64:               └→タブコードが見つかったら,xline[ ]の対応する箇所にもタブコードを埋めこむ
 65:     if(i != 0)………カラム数が0でない(検索文字列が見つかった)場合のみ表示
 66:         {
 67:         printf("%5d: %s", lnum+1, line);………文字列に行番号を付けて表示
 68:         printf("       %s¥n", xline);    ………見つかった文字の位置に印の付いた
 69:         }                                       配列xlineを表示
 70:     }
 71:
 72: /*
 73:     Search the strings. ………search関数は,別のプログラムでも使うことを
 74:                              考慮してコメントを付けておいた
 75:             n  =  search(str0,strl)
 76:
 77:                 return  = n  Good
 78:                 return  = -1 No Good
 79:
 80:             str0[(bigger)]
 81:             strl[(smaller)]
 82:
 83:     The function search the strings strl, within str0.
 84: */
 85:
 86: int search(ix, str0, strl)……指定された文字列を探す関数
 87:     int     ix;…………………検索を開始するカラム位置
 88:     char    str0[]; ………読み込まれた文字列
 89:     char    strl[]; ………検索文字列
 90:     {                                     ┌(flag=−1…見つからない
 91:     int   i;                              └(flag=n……n文字目に見つかった
 92:     int   j;   ………文字列が見つかったかどうか,見つかった場合何文字で見つかったかが入るフラグ
 93:     int   flag;
 94:            ………………読み込んだ文字列の検索していないカラム数が検索文字列より小さい場合,−1を返す
 95:     if((strlen(str0) - ix) < strlen(strl)) return(-1);
 96:     if(0 == strlen(strl))                  return(-1);
 97:            ………検索文字列の長さが0ならば−1を返す
 98:     for(i = ix ; str0[i] != 0 ; ++i)  ……………………現在のカラム数から文字列の
 99:         {                                            最後まで繰り返す
100:         for(j = 0 ; strl[j] != 0 ; ++j)  ……………検索文字列の1文字目から
101:             {                                        最後まで繰り返す
102:             if(str0[i + j] != strl[j])
103:                 {                    ……文字列の一致を調べる
104:                 flag = (-1);  } 一致しない場合−1をflagに代入し
105:                 break;        } ループを抜ける
106:                 }
107:             else
108:                 flag = i; ………一致した時の最初のカラム数を
109:             }                   flagにとっておく
110:         if((-1) != flag)………検索文字列が見つかった場合には,
111:             break;              for文を抜ける
112:         }
113:     return(flag); ………関数の返値としてflagの値を渡す
114:     }
115:
```

```
A>cgrep↵

***** Input File-name              : cgerp.c↵ ………ファイル名を入力
***** Input strings for search     : str↵ ………検索文字列を入力

 15:      char    str_in[64];
                  ^ ………文字列の次の行に,検索結果を表示
 29:      printf("***** Input strings for search   : ");
                                  ^
 30:      gets(str_in);
               ^
 36:          dspline(line_in, str_in, x);
                                ^
 42:
    10:45 1986   Page 4

100:          for(j = 0 ; strl[j] != 0 ; ++j)
                          ^
102:              if(str0[i + j] != strl[j])
                     ^             ^
A>
```

MSX-C, LSI C…「22: gets(fname, 64);」に変更
22行目の後に「fname[strlen(fname)-1] = 0x00;」を追加
「30: gets(str_in, 64);」に変更
30行目の後に「str_in[strlen(str_in)-1] = 0x00;」を追加
BDS C …………「8: #include <bdscio.h>」に変更
「18: FILE ibuf;」に変更
「23: if((-1) == fopen(fname, ibuf))」に変更
「35: if(NULL == fgets(line_in, ibuf)) break;」に変更
「39: fclose(ibuf);」に変更
RUN/C …………8行目を削除
「23: if((fp = fopen(fname, "r")) == NULL)」に変更
「35: if(fgets(line_in, 255, fp) == NULL) break;」に変更

リスト 6-4 cgrepプログラムと実行結果

■プログラムの解説

このプログラムの重要な点について，以下に解説します．細かな点は，リストのなかに示してありますので参照してください．

10行目の#define文は，プログラム中の"HTAB"という単語を"0x09"(タブのキャラクタコード)に置き換えることを指定するプリプロセッサです．このようにキャラクタコードなどは，わかりやすい単語に置き換えておくと，ソースファイルが読みやすくなります．

main 関数は，ファイル名と検索文字列を標準入力(キーボード)から取り込みファイルのオープンを行います．22 行目では，これまで使っていた scanf 関数の代わりに 1 行入力用の標準関数である〝gets〞が使ってあります．以下にその仕様を示します(**表 6-2**)．

書　式	gets(buff)	
パラメータ／返値	データ型	解　説
パラメータ1 (buff)	char型のポインタ ex) char ＊buff	標準入力から取り込んだデータを保存する変数buffのポインタ
関数の返値 (return value)	char型のポインタ	NULL＝エラー NULL≒読み込み成功
使用例	gets(buffer); 標準入力(キーボード)から，1行(キャリッジリターン)までを取り込み，変数buffer に保存	

表 6-2　gets関数の仕様

もう 1 つこのプログラムには，新しい関数が出てきます．50 行目にある strlen 関数がそれで，以降の行でも随所に出てきます．〝strlen〞は〝string length〞を意味し，渡された文字列の長さを調べます．ここで与える文字列は，「C ストリング」である点に注意してください．**表 6-3** にこの関数の仕様を示します．

書　式	strlen(string)	
パラメータ／返値	データ型	解　説
パラメータ1 (string)	char型のポインタ ex) char ＊string	長さ(バイト数)を調べる 文字列(Cストリング)の入った変数のポインタ
関数の返値 (return value)	int型の実値	パラメータ1で渡した変数の 長さ(バイト数)
使用例	length＝strlen(string1); string1に入っている文字列の長さを調べてlengthに返す．string1はCストリングであること	

表 6-3　strlen関数の仕様

さて，このプログラムの中心は，86行目から始まるsearch関数です．この関数は，ある文字列のなかから指定した文字列を捜し出す働きをします．そして，何カラム目で文字列が見つかったかを呼ばれた関数に返します．

ここでむずかしいのは，for文で始まる2つのループの役割です．まず外側のfor文のカウンタ〝i″は常に現在検索中の文字列の先頭を示しています．ここで，カウンタの初期値ixは親の関数から渡され，文字列のどこから検索を開始するかが決められます．こうすることで，1行のなかでいくつも該当する文字列がある場合の対処をしています．

内側のfor文は，実際に2つの文字列の比較を行います．カウンタ〝j″は，検索すべき文字列の何文字目を調べているかを表しています．このループでは，1つずつカウンタを進めながら文字が一致するかどうかを見ています．ここで一致しないことがわかると，内側のループを抜けて外側のループに

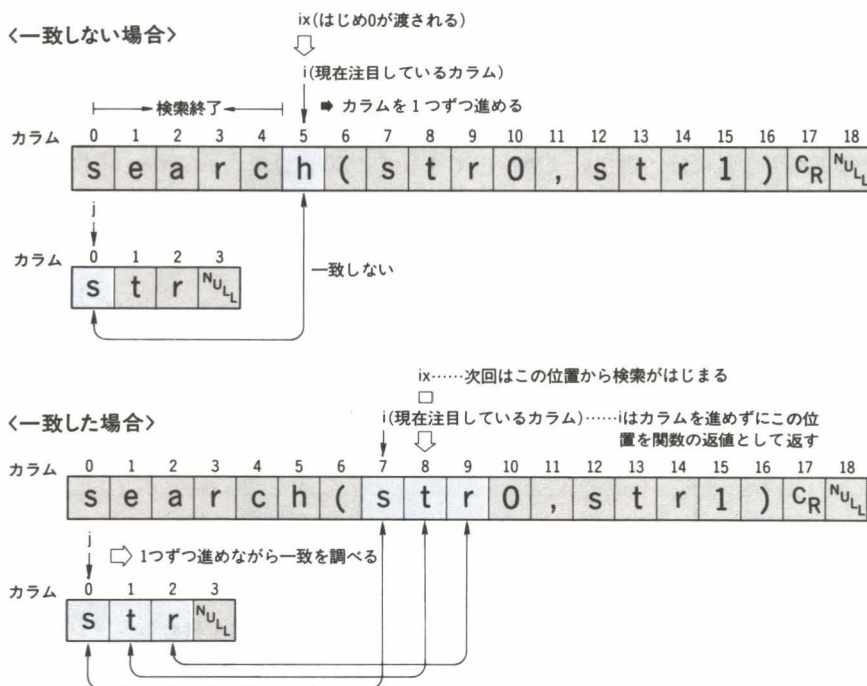



図 6-14　search関数の動作

戻り，同じ文字列が見つかった場合は一致した最初のカラム位置を持ってこの関数を抜け，呼ばれた関数に戻ります．

以上のことを**図 6-14** に整理しておきます．

検索結果を画面に表示する dspline 関数についても解説しておきます．ここでのポイントは，56行目から始まる while 文です．このループでは，読み込んだ行と検索文字列，検索を始めるカラム位置を search 関数に渡し，関数の返値として結果を受け取っています．まず，ループに入る前の 54 行目で変数が初期化されますから，最初は行の先頭から検索が始まります．ここで一致しなければ，search 関数から〝−1〟が返るので，このループを抜け，また 67 行目から始まる printf 関数も実行されません．一致した場合は該当するカラムの位置が返り，結果を表示する配列の同じ位置に〝^〟を書き込みます．そして，このループを抜けずに検索位置を 1 カラム進めて，さらに該当する文字列がないかどうか調べています．

図 6-15 に dspline 関数の動作をまとめます．

図 6-15　dspline関数の動作

これまでの説明以外の部分でとくにむずかしいところはありません．後は，プログラムリストを参照して自分で解析してみてください．

なお，C 言語によっては，search 関数を作らなくとも同じ機能を持った「index」という標準関数が用意されているものもあります．index 関数が備わっている C 言語をお持ちの方は，index 関数に置き換えたプログラムを作ってみてください（機能はほぼ同じですが，書式や使い方は違います）．

【この章のポイント】

1．コンパイルとリンクの必要性と意味

2．オブジェクトファイルは，メモリ上のどこにでも置くことができる

3．ライブラリは，標準関数のオブジェクトファイルの集まりである

4．エラーが出たときの対処の方法

・Warning(警告)とError(間違い)の違いに気をつける

・エラーメッセージの先頭の行から見る

・エラーメッセージが表示された行と，その1つ前の行を見る

・かっこ，セミコロン，カンマなど区切り記号が抜けていないかどうかに注意する

5．プログラムを書く前に，仕様書を作ること

[練習問題]

1．kakkoまたはcgrepプログラムのどこか1カ所の「{」もしくは「;」を取り去って，故意にエラーを作り，そのエラーメッセージを見ること

2．キーボードから入力した文字をファイルにするプログラムを書きなさい

(解答は220ページ)

APPENDIX

各社のC言語の紹介

MS-DOS

Microsoft C Compiler Ver . 3.0

発売元：マイクロソフト㈱

〒102 東京都千代田区三番町6-2 三番町弥生館 Tel 03-221-7074

価 格：98,000円（昭和61年11月1日現在）

■ディスクの用意

● ドライブAの内容

各自でエディタを用意して以下に追加する．なお，640KBドライブの場合はlibディレクトリをドライブBに移す．

```
|-- COMMAND.COM ………MS-DOSのコマンドプロセッサ
|-- CONFIG.SYS   } 以下で示すように作成
|-- AUTOEXEC.BAT }
|-- P0.EXE }
|-- P1.EXE } MSC.EXEに呼ばれるコンパイラ群
|-- P2.EXE }
|-- P3.EXE }
|-- CL.EXE
|-- MSC.EXE ………コンパイラ本体
|-- EXEMOD.EXE  }
|-- EXEPACK.EXE } 各種ユーティリティ
|-- LIB.EXE     }
|-- LINK.EXE ………リンカ
|                        |-- ASSERT.H
|                        |-- CONIO.H
|                        |-- CTYPE.H
|                        |-- DIRECT.H
|                        |-- DOS.H
|                        |-- ERRNO.H
|                        |-- FCNTL.H
|                        |-- IO.H
|                        |-- MALLOC.H
|                        |-- MATH.H
|-- INCLUDE -------------|-- MEMORY.H
|       :                |-- PROCESS.H
|       :                |-- SEARCH.H
|  プログラムに取り込んで    |-- SETJMP.H
|  使うヘッダファイル群      |-- SHARE.H
|                        |-- SIGNAL.H
|                        |-- STDIO.H
|                        |-- STDLIB.H
|                        |-- STRING.H
|                        |-- TIME.H
|                        |-- V2TOV3.H
|                |-- SLIBC.LIB
|                |-- MLIBC.LIB
|                |-- LLIBC.LIB
|                |-- SLIBFP.LIB
|                |-- MLIBFP.LIB
|                |-- LLIBFP.LIB
|                |-- SLIBFA.LIB
|                |-- MLIBFA.LIB
|                |-- LLIBFA.LIB
|                |-- EM.LIB
|-- LIB ---------|-- 87.LIB
|     :          |-- EMOEM.ASM
|     :          |-- BINMODE.OBJ
| メーカーから供給 |-- SSETARGV.OBJ
| されたライブラリ  |-- LSETARGV.OBJ
| 群              |-- MSETARGV.OBJ
|                |-- SVARSTCK.OBJ
|                |-- MVARSTCK.OBJ
|                |-- LVARSTCK.OBJ
|                |-- PC98.OBJ
|
|-- CC.BAT ………以下で示すように作成する
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.C …………C言語のソースファイル
|-- TEST.OBJ ………オブジェクトファイル(中間ファイル)
|-- TEST.EXE ………実行形式のファイル
```

■ MS-DOS の設定

● CONFIG.SYS の設定

```
files = 20
buffers = 20
break = on
shell = a:¥command.com a:¥ /p
```

● AUTOEXEC.BAT の設定

```
echo off
path = a:¥;b:¥
set INCLUDE = a:¥include }
set LIB = a:¥lib         } ………コンパイラが使用するディレクトリの設定
set TMP = b:¥            }
```

■コンパイル

● CC.BAT の内容

```
msc %1.c, %1.obj;
link %1.obj, %1.exe, NUL, em.lib, slibfp.lib, slibc.lib
```

●コンパイルの方法

```
A>cc b:test⏎ ………ドライブBにあるソースファイル"test.c"をコンパイルする.「cc.bat」を使用する場合は,
                  拡張子を指定しない
A>msc b:test.c, b:test.obj;
Microsoft C Compiler Version 3.00
(C)Copyright Microsoft Corp 1984 1985

A>link b:test.obj, b:test.exe, NUL, em.lib, slibfp.lib, slibc.lib

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

A>test⏎ ………プログラムの実行
```

■補足

コンパイルは，付属の cl **コマンド**でも行える．以下にその方法を示す．

```
B>cl test.c⏎ ………clコマンドはドライブBから起動する.また,拡張子「.c」はかならず指定すること
Microsoft C Compiler Version 3.00
(C)Copyright Microsoft Corp 1984 1985
test.c

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

Object Modules [.OBJ]: TEST.OBJ
Run File [TEST.EXE]: TEST.EXE/NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]: ;

B>test⏎
```

MS-DOS

Lattice C　Ver . 3.0

発売元：㈱ライフボート

〒101　東京都千代田区神田錦町 3-6　Tel 03-293-4711

価　格：PC-9801 版…98,000 円（昭和 61 年 3 月 1 日現在）†

■ディスクの用意

● ドライブAの内容

各自でエディタを用意して以下に追加する．またリンカ(LINK . EXE)は，MS-DOSのシステム・ディスクからコピーする．

```
|-- COMMAND.COM ………MS-DOSのコマンドプロセッサ                     |                 |-- ASSERT.H
|-- CONFIG.SYS  } 以下で示すように作成                               |                 |-- CTYPE.H
|-- AUTOEXEC.BAT}                                                    |                 |-- DOS.H
|-- LC.EXE ………コンパイラ本体                                        |                 |-- ERROR.H
|-- LC1.EXE } LC.EXEに呼ばれるコンパイラ群                          |                 |-- FCNTL.H
|-- LC2.EXE }                                                        |                 |-- IOS1.H
|-- OMD.EXE                                                          |                 |-- LIMITS.H
|-- OML.EXE                                                          |                 |-- LOCKING.H
|-- LINK.EXE ………リンカ(MS-DOSのシステム・ディスクからコピーする)    |-- INCLUDE --|-- MATH.H
|-- CC.BAT ………以下で示すように作成                                  |      :          |-- SETJMP.H
|           |-- C.OBJ     }                                          |      :          |-- SIGNAL.H
|           |-- _MAIN.OBJ }                                          |      :          |-- STDIO.H
|           |-- NDP.OBJ   }                                          | ヘッダファイル群 |-- STDLIB.H
|-- LIB --|-- LC.LIB      } ライブラリ群                             |                 |-- STRING.H
|           |-- LCM.LIB   }                                          |                 |-- TIME.H
|           |-- NONDP.LIB }                                          |                 |-- UNLSTD.H
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.C   ………C言語のソースファイル
|-- TEST.OBJ ……オブジェクトファイル
|-- TEST.EXE ……実行形式のファイル
```

† 昭和61年11月 1 日現在，バージョンが3.1 にアップされている

■ MS-DOS の設定

● CONFIG . SYS の設定

```
files = 20
buffers = 20
break = on
shell = a:¥command.com a:¥ /p
```

● AUTOEXEC . BAT の設定

```
echo off
path = a:¥;b:¥
set INCLUDE=a:¥include¥ ………ヘッダファイルを捜すディレクトリの設定
```

■コンパイル

● CC . BAT の内容

```
lc -ms -ccdmsuw %1.c
link ¥lib¥c+%1.obj, %1.exe, NUL, ¥lib¥lc
```

●コンパイルの方法

```
A>cc b:test↵ ………ドライブBにあるソースファイル"test.c"をコンパイルする. 拡張子「.c」は指定しない点に注意

A>lc -ms -ccdmsuw b:test.c
Lattice MS-DOS C Compiler, Version 3.00
Copyright (C) 1985 Lattice, Inc.  All rights reserved.

Compiling b:TEST.C
Module size P=000E D=0011 U=0000

Total files: 1, Successful compilations: 1

A>link ¥lib¥c+b:test.obj, b:test.exe, NUL, ¥lib¥lc

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985


A>test↵ ………プログラムの実行
```

■補足

ドライブAのディスクは, 付属の install コマンドで作成した(スモールモデル用)のディスクを基に作成している

MS-DOS

PM-MWC 86　Ver. 2.15

発売元：パーソナルメディア㈱

〒141　東京都品川区西五反田2-12-19　五反田NNビル 10F　Tel 03-495-6241

価　格：フロッピーディスク版…189,000円，ハードディスク対応版…239,000円

(昭和61年3月1日現在)

■ディスクの用意

● ドライブAの内容

各自でエディタを用意して以下に追加する．またリンカ(LINK.EXE)は，MS-DOSのシステム・ディスクからコピーする．

```
☆|-- COMMAND.COM ………MS-DOSのコマンドプロセッサ
☆|-- CPP.EXE
 |-- CC0.EXE    ┐
 |-- CC1.EXE    │
 |-- CC2.EXE    │
 |-- LD.EXE     │ MWC独自のフォーマット
 |-- LIBC.OLB   │ のオブジェクトファイル
 |-- LIBM.OLB   │ を作成するコンパイラと
 |-- CRTS0.O    │ ライブラリ
 |-- F.O        │
 |-- NS.O       ┘
☆|-- CC0X.EXE   ┐
☆|-- CC1X.EXE   │ コンパイラ群
☆|-- CC2X.EXE   ┘
☆|-- LIBCXS.LIB ┐
☆|-- LIBMXS.LIB │
☆|-- SCRTS0.OBJ │ ライブラリ群
☆|-- FXS.OBJ    │
☆|-- NSXS.OBJ   ┘

☆|-- ASSERT.H   ┐
☆|-- CTYPE.H    │
☆|-- DOS.H      │ ヘッダファイル群
☆|-- MATH.H     │
☆|-- SETJMP.H   │
☆|-- STDIO.H    ┘
 |-- NM.EXE     ┐
 |-- SIZE.EXE   │
 |-- STRIP.EXE  │
 |-- LB.EXE     │
 |-- AS.EXE     │ 各種ユーティリティ
 |-- CSD.EXE    │ プログラム
 |-- FACTOR.EXE │
 |-- INFL.EXE   │
 |-- MOVEALL.EXE┘
☆|-- CONFIG.SYS   ┐ 以下で示すように作成する
☆|-- AUTOEXEC.BAT ┘
☆|-- CC.EXE   ………コンパイル用のコマンド
☆|-- LINK.EXE ………リンカ(MS-DOSのシステム・ディスクからコピーする)

 |-- FACTOR.C
 |-- SQUARE.C
 |-- BANNER.C
 |-- HELLO.C
 |-- ATOD.C
 |-- INFL.C
 |-- FDIR.C
 |-- CSDEVAL.HLP
 |-- CSDTRACE.HLP
 |-- CSDEDIT.HLP
 |-- CSDHELP.HLP
 |-- CSDRUN.HLP
 |-- CSDFIND.HLP
 |-- CSDSELEC.HLP
```

☆は640KBドライブの人が入れておくファイル

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.C   ………C言語のソースファイル
|-- TEST.OBJ ………オブジェクトファイル(中間ファイル)
|-- TEST.EXE ………実行形式のファイル
```

■ MS-DOS の設定

● CONFIG.SYS の設定

```
files = 20
buffers = 20
break = on
shell = a:¥command.com a:¥ /p
```

● AUTOEXEC.BAT の設定

```
echo off
path = a:¥;b:¥
set cchead=-xca:¥ -xtb:¥ -xob:¥ -vsmall ……… コンパイル時のオプションの設定
```

■コンパイル

PM-MWC に付属しているコンパイル用のコマンド「CC.EXE」を使ってコンパイルを行う.

●コンパイルの方法

```
A>cc b:test.c ↵ ………ドライブBにあるソースファイル"test.c"をコンパイルする.拡張子「.c」はかならず指定する
PM-MWC86 is distributed by Personal Media Corporation, Tokyo Japan
MWC-86 is a trademark of Mark Williams Company, Chicago, Illinois USA
MWC-86 Version 2.1.5 (c) 1982, 1983, 1984, 1985
by Mark Williams Company, Chicago.  For use with MS-DOS Version 2.0.

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984, 1985

Object Modules [.OBJ]: A:¥SCRTS0+
Object Modules [.OBJ]: B:¥TEST
Run File [A:SCRTS0.EXE]: B:¥TEST.EXE
List File [NUL.MAP]: NUL
Libraries [.LIB]: A:¥LIBCXS

A>test ↵ ………プログラムの実行
```

■補足

PM-MWC では,「LINK.EXE」ではなく付属のリンカ「LD.EXE」を使用してリンクを行うこともできる.ただし,その場合にはオブジェクトファイルが独自のフォーマットになるので本書では使っていない.

MS-DOS　CP/M-86
HI-TECH C　　Ver. 3.05
発売元：㈱サザンパシフィック 〒220　神奈川県横浜市西区南幸2-16-20　三和横浜ビル　Tel 045-314-9514 **価　格**：47,500円．なおCP/M-86版はVer. 3.04（昭和62年1月1日現在）

■ディスクの用意

● ドライブAの内容

各自でエディタを用意して以下に追加する．

```
|-- COMMAND.COM ………MS-DOSのコマンドプロセッサ
|-- CONFIG.SYS  } 以下で示すように作成する
|-- AUTOEXEC.BAT}
|-- C.EXE ………コンパイル用のコマンド
|            |-- CPP.EXE       }
|            |-- P1.EXE        }
|            |-- SCG86.EXE     }
|            |-- OPT86.EXE     } コンパイラ群
|            |-- AS86.EXE      }
|            |-- OBJTOHEX.EXE  }
|-- HITECH --|-- LINK.EXE ………リンカ
|            |-- LIBSC.LIB     }
|            |-- LIBSF.LIB     } ライブラリ群
|            |-- SFRTDOS.OBJ   }
|            |-- SCRTDOS.OBJ   }
|
|                 |-- CTYPE.H
|                 |-- SETJMP.H
|                 |-- SIGNAL.H
|                 |-- STDIO.H
|                 |-- IOCTL.H
|-- INCLUDE --|-- MATH.H
|      :          |-- TIME.H
|  ヘッダファイル群 |-- LIMITS.H
|                 |-- STDEFS.H
|                 |-- ASSERT.H
|-- CC.BAT ………以下で示すように作成する
|-- LIBR.EXE      }
|-- DUMP.EXE      }
|-- DEBUG86.EXE   } 各種ユーティリティ・プログラム
|-- DEHUFF.EXE    }
|-- LCG86.EXE     }
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.C   ………C言語のソースファイル
|-- TEST.OBJ ………オブジェクトファイル(中間ファイル)
|-- TEST.EXE ………実行形式のファイル
```

■ MS-DOSの設定

● CONFIG.SYSの設定

```
files = 20
buffers = 20
break = on
shell = a:¥command.com a:¥ /p
```

● AUTOEXEC.BATの設定

```
echo off
path = a:¥;b:¥
set INCLUDE=a:¥include¥ ……ヘッダファイルを捜すディレクトリの設定
set LIB=a:¥hitech¥ ……ライブラリを捜すディレクトリの設定
```

■コンパイル

● CC.BAT の内容

```
c -v -i%INCLUDE% %1 %LIB%libsc.lib
```

●コンパイルの方法

```
B>cc test.c↵  ………ドライブBのソースファイル"test.c"をコンパイルする．HI-TECH Cはドライブ B からコマンド
                  を起動する．また，拡張子「.c」はかならず指定すること
B>c -v -ia:¥include¥ test.c a:¥hitech¥libsc.lib
HI-TECH C COMPILER (MS-DOS) V2.0
Copyright (C) 1984 HI-TECH SOFTWARE
a:¥hitech¥cpp.exe -DDOS -DHI_TECH_C -Di8086 -Ia:¥include¥ -Ia:¥hitech
a:¥hitech¥p1.exe $ctmp1.$$$ $ctmp2.$$$
a:¥hitech¥scg86.exe $ctmp2.$$$ $ctmp1.$$$
a:¥hitech¥as86.exe -q -n -oTEST.OBJ $ctmp1.$$$
a:¥hitech¥link.exe -z -ptext,data=0/,bss -o1.obj a:¥hitech¥scrtdos.obj T
a:¥hitech¥objtohex.exe -e 1.obj TEST.EXE
del 1.obj
del $ctmp1.$$$
del $ctmp2.$$$

B>test↵  ………プログラムの実行
```

■ CP/M-86 の場合の設定とコンパイル

●ドライブAの内容

階層ディレクトリは使用できないので，すべてのファイルをルートディレクトリに置く．

●CC.SUB の内容

```
c -v -ia: $1 a:libsc.lib
```

●コンパイルの方法

```
A>submit cc b:test.c↵
            ⋮
B>test↵
```

なお，CP/M-86 版は，スモールモデルのみがサポートされている．

MS-DOS　CP/M-86

DeSmet C　Ver. 2.4

発売元：ソフトウェアインターナショナル㈱
〒107　東京都港区南青山2-9-28　Tel 03-479-7151
価　格：46,000円（昭和61年3月1日現在）†

■ディスクの用意

● ドライブAの内容

DeSmet Cには，スクリーン・エディタが付属している．

```
|-- COMMAND.COM  ………MS-DOSのコマンドプロセッサ
|-- CONFIG.SYS   } 以下で示すように作成
|-- AUTOEXEC.BAT }
|-- C88.EXE  ………コンパイラ
|-- GEN.EXE
|-- ASM88.EXE
|-- BIND.EXE ……リンカ
|-- SEE.EXE  ……スクリーン・エディタ
|-- CC.BAT  ………以下で示すように作成
|
|                     |-- STDIO.H  } ヘッダファイル
|-- INCLUDE --|-- MATH.H   }
|
|           |-- CSTDIO.S   } ライブラリ
|-- LIB --|-- CSTDIO7.S  }
|
|-- CLIST.EXE  }
|-- LIB88.EXE  } 各種ユーティリティ・プログラム
|-- RAM.COM    }
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.c    ………C言語のソースファイル
|-- TEST.O    ………オブジェクトファイル（中間ファイル）
|-- TEST.EXE  ………実行形式のファイル
```

■ MS-DOSの設定

● CONFIG.SYSの設定

```
files = 20
buffers = 20
break = on
shell = a:¥command.com a:¥ /p
```

† 昭和61年11月1日現在，バージョンが2.5にアップされている

● AUTOEXEC.BAT の設定

```
echo off
path = a:¥;b:¥
set INCLUDE=a:¥include¥  ┐
set LIB=a:¥lib¥          ├ コンパイラが使用するディレクトリの設定
set TMP=b                ┘
```

コンパイル

● CC.BAT の用意

```
c88 %1 i%INCLUDE% t%TMP%
bind %1 -l%LIB%
```

● コンパイルの方法

```
A>cc b:test↵ ………ドライブBにあるソースファイル"test.c"をコンパイルする. 拡張子「.c」は指定しない

A>c88 b:test ia:¥include¥ tb
C88 Compiler    V2.41    (c) Mark DeSmet, 1982,83,84,85
end of C88        000E code   0006 data       1% utilization
A>bind b:test -la:¥lib¥
Binder for C88 and ASM88    V1.8    (c) Mark DeSmet, 1982,83,84,85
end of BIND        22% utilization

A>test↵ ………プログラムの実行
```

CP/M-86 の場合の設定とコンパイル

● ドライブAの内容

階層ディレクトリは使用できないので，すべてのファイルをルートディレクトリに置く.

● CC.SUB の内容

```
c88 $1 ia tb
bind $1 -la
```

● コンパイルの方法

```
A>submit cc b:test↵
        :
        :
B>test↵
```

MS-DOS

RUN/C　Ver. 2.0

発売元：㈱ライフボート

〒101　東京都千代田区神田錦町3-6　Tel 03-293-4711

価　格：33,000円（昭和61年11月1日現在）

■ディスクの用意

●ドライブAの内容

RUN/Cには，ライン・エディタが内蔵されている．

```
|-- COMMAND.COM  ………MS-DOSのコマンドプロセッサ
|-- CONFIG.SYS   } 以下で示すように作成
|-- AUTOEXEC.BAT }
|-- RC.EXE  ………RUN/Cインタプリタ本体
|-- INTRPT.H
```

■MS-DOSの設定

● CONFIG.SYSの設定

```
files = 20
buffers = 20
break = on
shell = a:¥command.com a:¥ /p
```

● AUTOEXEC.BATの設定

```
echo off
path = a:¥;b:¥
rc -l1000 -d20 ……起動時にRUN/Cが立ち上がるように設定
```

■インタープリタの実行方法

```
RUN/C, The C Interpreter, Version 1.21
by Stephen Walton with Peter Brooks
Copyright (C) 1984, 1985 by Age of Reason Co. All rights reserved.
241456 bytes free
              *RUN/Cは，インタープリタ型のC言語であり，BASICと同じように
               エディタのなかでプログラムの作成，実行などを行う
Ok
load "b:test"↵ ………BASICと同じようにloadコマンドでプログラムを読み込む

Loading line 4

Ok
list↵ ………プログラムを画面上に表示
    1:main()
    2:      {
    3:      printf("This is a test.¥n");
    4:      }
    └──────→ 行番号を付けて表示する
Ok
run↵ ………プログラムの実行
This is a test.

Ok
```

MSX-DOS

MSX-C　Ver. 1.00

発売元：㈱アスキー
〒107　東京都港区南青山6-11-1　スリーエフ南青山ビル　Tel 03-486-8080
価　格：98,000円（昭和61年3月1日現在）

■ディスクの用意

● ドライブAの内容

MSX-Cには，スクリーン・エディタが付属している．

```
|-- COMMAND.COM ····MSX-DOSのコマンドプロセッサ
|-- MSXDOS.SYS ·······MSX-DOSのシステムファイル
|-- CF.COM   }
|-- FPC.COM  } コンパイラ群
|-- CG.COM   }
|-- M80,COM  }
|-- L80.COM ·········リンカ
|-- CREF80.COM }
|-- MX.COM     } ユーティリティ・プログラム
|-- LIB80.COM  }
|-- SCED.COM } スクリーン・エディタ
|-- SCED.HLP }
|-- C.BAT
|-- CC.BAT ·········以下で示すように作成
|-- STDIO.H    }
|-- BDOSFUNC.H } ヘッダファイル
|-- CPMFUNC.H  }
|-- CEND.REL }
|-- CK.REL   }
|-- CLIB.REL } ライブラリ
|-- CRUN.REL }
|-- LIB.TCO  }
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.C   ·········C言語のソースファイル
|-- TEST.REL ·········オブジェクトファイル(中間ファイル)
|-- TEST.COM ·········実行形式のファイル
```

■コンパイル

● CC.BATの用意

```
CF -F -T %1
FPC LIB %1
CG -K %1
M80 = %1.ASM·········MSX-C Ver1.1では「M80 = %1/Z」に変更
L80 CK,%1,CLIB/S,CRUN/S,CEND,%1/N/E:XMAIN
DEL %1.ASM
```

●コンパイルの方法

```
A>cc b:test⏎ ………ドライブBにあるソースファイル"test.c"をコンパイルする．拡張子「.c」を指定しない点に注意

A>CF -F -T B:TEST
MSX C ver 1.00  (parser)
Copyright (C) 1985 by ASCII Corporation
complete
A>FPC LIB B:TEST
MSX C function parameter checker ver 1.0
complete
A>CG -K B:TEST
MSX C ver 1.00  (code generator)
Copyright (C) 1985 by ASCII Corporation
main     .;
complete
A>M80 = B:TEST.ASM

No Fatal error(s)

A>L80 CK,B:TEST,CLIB/S,CRUN/S,CEND,B:TEST/N/E:XMAIN

MSX.L-80  1.00  01-Apr-85    (C)  1981, 1985 Microsoft

Data     0103     17EC     < 5865>

34533 Bytes Free
[0210    17EC       23]

A>DEL B:TEST.ASM
A>b:⏎ ………カレント・ドライブをドライブBに切り換える
B>test⏎ ……プログラムの実行
```

■ UNIX上のC言語とのおもな相違点

- float, long, double のデータ型をサポートしていない．
- 関数およびパラメータに対する自動的な宣言は行われない．

  本書では「CC.BAT」のなかでコンパイル・オプションを指定して型宣言を自動的に行うように設定している．本書を理解された方は，オプションを指定しなくともコンパイルが行えるようにプログラムを変更するとよい．

＊ UNIX上のCコンパイラについて

本書のプログラムは，UNIX上のC言語でも動作を確認している．UNIX上のコンパイル方法を以下に示す．

```
        ┌──── 実行形式のファイル名を指定するオプション
        │   ┌──── 実行形式のファイル名
        │   │      ┌──── ソースファイル名．拡張子「.c」を指定する
%cc -o test test.c⏎
```

CP/M-80

LSI C　Ver. 1.1A

発売元：エルエス・アイ ジャパン㈱

〒151 東京都渋谷区千駄ヶ谷1-8-14 Tel 03-478-0575

価　格：130,000円（昭和61年3月1日現在）†

■ディスクの用意

● ドライブAの内容

各自でエディタを用意して以下に追加する．また，アセンブラ(M80.COM)，リンカ(L80.COM)が付属していないので用意する(ともにMicrosoft社製)．

```
|-- CF.COM   ┐                |-- STDIO.H     ┐ヘッダファイル        |-- MX.COM ┐ ユーティリティ・
|-- CG.COM   │コンパイラ群     |-- BDOSFUNC.H  ┘                      |-- @.COM  │ プログラム
|-- FPC.COM  ┘                |-- M80.COM ……アセンブラ┐LSI Cには          |-- LS.COM ┘
|-- CRUN.REL ┐                |-- L80.COM ……リンカ    ┘付属していない
|-- CEND.REL │                |-- SUBMIT.COM ………CP/M-80のシステム・ディスクからコピーしておく
|-- CLIB.REL │ライブラリ群     |-- CC.SUB ………………以下で示すように作成
|-- LIB.TCO  ┘
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．

```
|-- TEST.C    ………C言語のソースファイル
|-- TEST.REL  ………オブジェクトファイル(中間ファイル)
|-- TEST.COM  ………実行形式のファイル
```

■コンパイル

● CC.SUBの用意

```
CF -F -T $1
FPC LIB $1
CG -K $1
M80 = $1.ASM
L80 CK,$1,CLIB/S,CRUN/S,CEND,$1/N/E:XMAIN
ERA $1.ASM
```

† 昭和61年11月1日現在，バージョンが2.1にアップされている

●コンパイルの方法

```
      ┌─サブミットファイル(.sub)の起動にはsubmitコマンドを用いる
A>submit cc b:test ↵ ………ドライブBにあるソースファイル"test.c"をコンパイルする．拡張子「.c」は指定しない

A>CF -F -T B:TEST
LSI C ver 1.43  (parser)
Copyright (C) 1983,1984 by LSI JAPAN Co.,Ltd.  [000304]
complete
A>FPC LIB B:TEST
LSI C function parameter checker ver 1.61
complete
A>CG -K B:TEST
LSI C ver 1.25  (code generator)
Copyright (C) 1983,1984 by LSI JAPAN Co.,Ltd.  [000304]
main     .;
complete
A>M80 = B:TEST.ASM

No Fatal error(s)

A>L80 CK,B:TEST,CLIB/S,CRUN/S,CEND,B:TEST/N/E:XMAIN

Link-80  3.34  14-Apr-81  Copyright (c) 1981 Microsoft

Data     0103     17EC     < 5865>

34533 Bytes Free
[0210   17EC        23]

A>ERA B:TEST.ASM
A>b: ↵ …………カレント・ドライブをドライブBに切り換える
B>test ↵ ………プログラムの実行
```

■ UNIX上のC言語とのおもな相違点

● float, long, double のデータ型をサポートしていない．

●関数およびパラメータに対する自動的な宣言は行われない．

本書では「CC.SUB」のなかでコンパイル・オプションを指定して型宣言を自動的に行うように設定している．本書を理解された方は，オプションを指定しなくともコンパイルが行えるようにプログラムを変更するとよい．

CP/M-80

BDS C Ver.1.5a

発売元:㈱ライフボート

〒101 東京都千代田区神田錦町3-6 Tel 03-293-4711

価 格:60,000円(昭和61年3月1日現在)

■ディスクの用意

●ドライブAの内容

各自でエディタを用意して以下に追加する.

```
|-- CC.COM     } コンパイラ
|-- CC2.COM    }
|-- CLINK.COM ……リンカ
|-- CLIB.COM
|-- C.CCC      }
|-- DEFF.CRL   } ライブラリ
|-- DEFF2.CRL  }
|-- BDSCIO.H   }
|-- DIO.H      } ヘッダファイル
|-- HARDWARE.H }
|-- SUBMIT.COM ………CP/M-80のシステム・ディスクからコピーしておく
|-- CC.SUB  ………
|-- LONG.C
|-- FLOAT.C
|-- WILDEXP.C
               以下で示すように作成
```

●ドライブBの内容

C言語のソースファイル,および実行形式のファイルをいれる.

```
|-- TEST.C   ……C言語のソースファイル
|-- TEST.CRL ……オブジェクトファイル
|-- TEST.COM ……実行形式のファイル
```

■コンパイル

●CC.SUBの用意

```
CC $1.c -X
CLINK $1.crl
```

●コンパイルの方法

```
A>submit cc b:test ↵  ………ドライブBにあるソースファイル"test.c"をコンパイルする.
                            拡張子「.c」は指定しない
  (サブミットファイル(.sub)の起動にはsubmitコマンドを用いる)
A>CC B:TEST.C -X
BD Software C Compiler v1.50a (part I)
  31K elbowroom
BD Software C Compiler v1.50 (part II)
  28K to spare
A>CLINK B:TEST.CRL
BD Software C Linker   v1.50
Linkage complete
  39K left over
A>b: ↵     ………カレント・ドライブをドライブBに切り換える
B>test ↵   ………プログラムの実行
```

■UNIX上のC言語とのおもな相違点

●変数のデータ型は, int, unsigned, charのみがサポートされる.

●記憶クラスは, auto変数とグローバル変数のみがサポートされる.

●通常「stdio.h」に定義されている標準入出力関数は,BDS Cの場合「bdscio.h」に定義されている.

CP/M-80

HI-TECH C　Z80　Ver. 3.05

発売元：㈱サザンパシフィック

〒220　神奈川県横浜市西区南幸2-16-20　三和横浜ビル　Tel 045-314-9514

価　格：42,500円（昭和62年1月1日現在）

■ディスクの用意

● ドライブAの内容

```
|-- C.COM ………コンパイル用のコマンド
|-- CPP.COM   ┐
|-- P1.COM    │
|-- CGEN.COM  │
|-- OPTIM.COM │ コンパイラ群
|-- ZAS.COM   │
|-- $EXEC.COM ┘
|-- LINK.COM ………リンカ
|-- OBJTOHEX.COM
|-- LIBC.LIB   ┐
|-- CRTCPM.OBJ │ ライブラリ
|-- RRTCPM.OBJ ┘
|-- STDIO.H  ┐
|-- CPM.H    │
|-- CTYPE.H  │ ヘッダ
|-- SIGNAL.H │ ファイル群
|-- SETJMP.H │
|-- EXEC.H   ┘
```

● ドライブBの内容

C言語のソースファイル，および実行形式のファイルを入れる．320KBタイプのディスクではドライブAにエディタが入らないのでドライブBに置く．また，ライブラリの一部もドライブBに入れておく．

```
|-- LIBF.LIB ………ライブラリ
|-- TEST.C   ………C言語のソースファイル(中間ファイルは，コンパイル用のコマンドが自動的に消去する)
|-- TEST.COM ………実行形式のファイル
```

■コンパイル

HI-TECH Cに付属しているコンパイル用のコマンド「C.COM」を使ってコンパイルを行う．

```
B>a:c test.c↵ ………C言語のソースファイル"test.c"をコンパイルする．HI-TECH CではドライブBから
HI-TECH C COMPILER V1.4        コマンドを起動する．また拡張子「.c」はかならず指定する
Copyright (C) 1984 HI-TECH SOFTWARE

B>test↵ ………プログラムの実行
```

キャラクタコード表

このコード表は，キャラクタコードを16進数で分類している．

たとえば，16進数で41は，「A」という文字を表す．また( )のなかは8進数でのコードを表す．

下位＼上位	0	1	2	3	4	5	6	7
0	NULL (0)	DLE (20)	スペース (40)	0 (60)	@ (100)	P (120)	` (140)	p (160)
1	SOH (01)	DC1 (21)	! (41)	1 (61)	A (101)	Q (121)	a (141)	q (161)
2	STX (02)	DC2 (22)	" (42)	2 (62)	B (102)	R (122)	b (142)	r (162)
3	ETX (03)	DC3 (23)	# (43)	3 (63)	C (103)	S (123)	c (143)	s (163)
4	EOT (04)	DC4 (24)	$ (44)	4 (64)	D (104)	T (124)	d (144)	t (164)
5	ENQ (05)	NAK (25)	% (45)	5 (65)	E (105)	U (125)	e (145)	u (165)
6	ACK (06)	SYN (26)	& (46)	6 (66)	F (106)	V (126)	f (146)	v (166)
7	BEL (07)	ETB (27)	' (47)	7 (67)	G (107)	W (127)	g (147)	w (167)
8	BS (10)	CAN (30)	((50)	8 (70)	H (110)	X (130)	h (150)	x (170)
9	HT (11)	EM (31)	) (51)	9 (71)	I (111)	Y (131)	i (151)	y (171)
A	LF (12)	SUB (32)	* (52)	: (72)	J (112)	Z (132)	j (152)	z (172)
B	VT (13)	ESC (33)	+ (53)	; (73)	K (113)	[(133)	k (153)	{ (173)
C	FF (14)	FS (34)	, (54)	< (74)	L (114)	¥ (134)	l (154)	¦ (174)
D	CR (15)	GS (35)	− (55)	= (75)	M (115)	] (135)	m (155)	} (175)
E	SO (16)	RS (36)	. (56)	> (76)	N (116)	^ (136)	n (156)	~ (176)
F	SI (17)	VS (37)	/ (57)	? (77)	O (117)	_ (137)	o (157)	DEL (177)

注意：
ASCIIコードでは「¥」が
「＼」(バックスラッシュ)になる

■ コントロールキャラクタの文字

画面上に表示されず，画面の制御などを行う文字をコントロールキャラクタという．

ここでは，C言語のプログラムでとくによく使われるものを示す．

16進数	8進数	記号	意　味
00	00	NULL	通常は文字として認識しない文字．C言語では文字列の最後にかならず入る
07	07	BEL	(Bell)ブザーを鳴らす
08	10	BS	(Back Space) 1文字前に戻る
09	11	HT	(Holizontal Tabulation)水平方向の次のタブ位置への移動
0A	12	LF	(Line Feed) 1行送る
0C	14	FF	(Form Feed)次のページへの移動(画面クリア)
0D	15	CR	(Carridge Return)行の先頭位置への移動
1B	33	ESC	(Escape)制御の拡張を行う

printf関数の使い方

書式	printf（表現指示文字列，引数1，引数2，……）;

〈表現指示文字列〉

(1)データ表現の型の指定

型	意　味	使用例	結　果
d	整数(int)を10進数で表す	int a = 123; printf("%d", a);	123
x	整数(int)を16進数で表す	int a = 123; printf("%x", a);	7b
o	整数(int)を8進数で表す	int a = 123; printf("%o", a);	173
u	整数を符号なしの10進数で表す	unsigned int a = 10; printf("%u", a);	10
s	文字列をそのまま表す	static char a[ ] ="ABC"; printf("%s", a);	ABC
c	1文字をそのまま表す	char a = '3'; printf("%c", a);	3
e	floatの数値を指数形式で表す	float a = 567.789; printf("%3.3e", a);	5.678e+02
f	floatの数値を実数形式で表す	float a = 567.789; printf("%3.3f", a);	567.789
g	上記eかfのうち短い表現を選んで表示する		

＊long型の数値を扱う場合はC言語によって異なるが,「d」を「D」もしくは「ld」とするのが一般的である.

(2)小数点以下の桁数／表示する桁数の指定

指　定	意　味	使　用　例	結　果
数値の場合	小数点以下の桁数の指定	float a = 1.23; printf("%2.3f", a);	␣1.230
文字列の場合	表示する文字数の指定	static char a [ ] = "ABCDE"; printf("%5.3s", a);	␣␣ABC

＊省略可能

(3)小数点または表示する桁数の区切り

＊小数点以下の桁数または表示する桁数の指定がない場合は，省略される

(4)データを表示する最大の桁数

指　定	使　用　例	結　果
データを表示する桁数を指定する（符号桁を含む）	int a = 123; printf("%5d", a);	␣␣123

＊省略可能

(5)いらない桁を0で埋めるかどうか（ゼロサプレス）の指定

指　定	意　味	使　用　例	結　果
0を指定	数値の入らない桁が0で埋まる	int a = 123; printf("%05d", a);	00123
指定を省略	数値の入らない桁が空白で埋まる	int a = 678; printf("%5d", a);	␣␣678

(6)右詰め，左詰めの指定

記 号	意　味	使　用　例	結　果
−	データの出力を左詰めにする	static char a[ ] = "ABC"; printf("%−5s", a);	ABC␣␣
+	データの出力を右詰めにする	static char a[ ] = "ABC"; printf("%+5s", a);	␣␣ABC

＊省略した場合は右詰めで表示される

〈表現指示文字列と引数の対応〉

置き換えたい順番に引数を並べる

> 注意：細かい部分はC言語によって異なるので，各マニュアルを確認すること
> 表現指示の文字は，scanf関数でもそのまま使うことができるが，引数の渡し方が違う
> 通常は引数としてのデータのポインタを渡す（第4章を参照）

標準関数一覧表

■一覧表の見方

この一覧表は，第5章で示した標準関数のなかでとくによく使用するものを整理してあります．この表の書式の見方を以下に示します．

<書式>

関数の返値(Return Value)がある場合はその型を表す(この場合はchar型のポインタ)

関数から返値を受け取りたい場合は，「char *a;」または「char a[ ];」などと宣言した変数で受け取る

```
char * fgets (b, n, fp) ;
```

引数の並び

[引数の型]……関数に渡す引数は以下の型で宣言する

char *b ; ……… char型のポインタ

ポインタの場合は，「char b[ ];」と宣言することもできるが，この表では「char *b;」で統一してある

int n ; …………… int型の実値(通常の変数)

FILE *fp ; ………構造体FILEへのポインタ

■入出力標準関数を使う前に

C言語の標準関数のなかでよく使う入出力関係の関数は，プログラムのはじめで以下のように[†]「stdio.h」というヘッダファイルを取り込んでから使用します(151ページ参照)．

```
#include <stdio.h>
```

■キャラクタコード分類関数

以下の関数は，[††]「ctype.h」というヘッダファイルに定義されています．使用にあたっては，

```
#include <ctype.h>
```

としてください．

† BDS Cは，「bdscio.h」となる．RUN/Cは，標準関数を使う際にヘッダファイルを取り込む必要はない

†† LSI C, MSX-Cでは「#include <stdio.h>」とする．DeSmet Cは，ヘッダファイルを取り込む必要はない

関数名	書　式	機　能	使　用　例
isalpha	int isalpha (c); int c;	アルファベットかどうかを調べ, アルファベットである≠0 アルファベットでない＝0 を返す	while (isalpha (c)) …… "c"がアルファベットである間, 続く実行文を繰り返す
isdigit	int isdigit (c); int c;	数値かどうかを調べ, 数値である≠0 数値でない＝0 を返す	while (isdigit (c)) …… "c"が数値(0〜9)である間, 続く実行文を繰り返す
islower	int islower (c); int c;	小文字かどうかを調べ, 小文字である≠0 小文字でない＝0 を返す	if (islower (c)) …… "c"が小文字(a〜z)であれば, 続く実行文を実行する
isspace	int isspace (c); int c;	空白文字かどうかを調べ, 空白文字である≠0 空白文字でない＝0 を返す	while (! isspace (c)) …… "c"が空白文字(0x09〜0x0d,0x20)でない間, 続く実行文を繰り返す
isupper	int isupper (c); int c;	大文字かどうかを調べ, 大文字である≠0 大文字でない＝0 を返す	if (isupper (c)) …… "c"が大文字(A〜Z)であれば, 続く実行文を実行する

■文字列操作関数

関数名	書　式	機　能	使　用　例
strcmp	int strcmp (s1, s2); char *s1; char *s2;	文字列 s1 と s2 のキャラクタコードを比較し, 以下の結果を返す s1<s2 の場合…0より小さい s1=s2 の場合…0 s1>s2 の場合…0より大きい	i = strcmp (s1, "test"); switch (i) …… 文字列 s1 と "test" を比較し, その結果を switch 文で判断する
strlen	int strlen (s1); char *s1;	文字列 s1 の長さを調べる	i = strlen (s1); 文字列 s1 の長さを変数 i に代入する
strcpy	char *strcpy (s1, s2); char *s1; char *s2;	文字列 s2 を文字列 s1 にコピーし, 文字列 s1 のポインタを返す	printf ("%s", strcpy (s1, "string")); "string"を文字列 s1 にコピーし, その文字列を表示する

注意：1．文字列 s1, s2 は，Cストリング(45ページ参照)であること

2．C言語によっては，「stdio.h」以外のヘッダファイルを取り込む必要がある．使用にあたっては，各マニュアルを参照のこと

■ファイル操作関数

関数名	書　式	機　能	使　用　例
fopen	FILE *fopen (file, mode); char *file; char *mode;	文字列 file で示すファイルを mode の状態でオープンする．返値は，構造体 FILE のポインタとなる．またオープンできない場合は NULL が返る	if (NULL==(fp=fopen (file, "r")))… 文字列 file で示すファイルを read モードでオープンし，その構造体 FILE へのポインタを fp に代入する．そして，ファイルがオープンできなければ続く実行文を実行する
fclose	int fclose (fp); FILE *fp;	fp で示すファイルをクローズする	fclose (fp); fp で示すファイルをクローズする
feof cf1	int feof (fp); FILE *fp;	ファイルの終わりかどうか調べ，以下の値を返す ファイルの終わり≠0 ファイルの終わりでない=0	if (feof (fp))…… fp で示すファイルが終わり(end of file)ならば，続く実行文を実行する
ferror cf1	int ferror (fp); FILE *fp;	ファイルの書き出し(読み込み)時にエラーが起きたかどうかを調べる エラーあり≠0 エラーなし=0	if (ferror (fp)) break; fp で示すファイルにエラーがあれば，制御文から抜ける

cf1:　BDS C, LSI C, MSX-C, DeSmet C には用意されていない

<ファイルをオープンする際の mode の指定>

mode	意　味
"r"	読み込み(read)用にオープンする
"w"	書き出し(write)用にオープンする．指定したファイルが存在した場合，ファイルの内容は失われる

このほかに次の mode が用意されている(C 言語によって異なる)．

mode	意　味
"a"	既存のファイルに追加書き出しするためにオープンする．オープンした時点でのファイルポインタはファイルの最後に設定される
"r+w"	読み込み/書き出しが共にできるようにファイルをオープンする

注意：1．テキストファイルとバイナリファイルでは，オープンする際のファイルの扱いが異なるオペレーティング・システムがある

2．ファイルをオープンする際の mode については，C 言語によって異なるのでマニュアルを参照すること

■バイト入出力関数

関数名	書　式	機　能	使　用　例
getc	int getc (fp); FILE *fp;	fp で示すファイルから 1 バイト読み込みファイルポインタを 1 つ進める	c = getc (fp); fp で示すファイルから 1 バイト(文字)読み込み，変数 c に代入する
putc	int putc (c, fp); int c; FILE *fp;	fp で示すファイルに変数 c の内容を書き出し，ファイルポインタを 1 つ進める	putc (c, fp); fp で示すファイルに変数 c の内容(1 文字)を書き出す

■文字列入出力関数

関数名	書　式	機　能	使　用　例
fgets cf1	char *fgets (b, n, fp); char *b; int n; FILE *fp;	fp で示すファイルから文字列を配列 b に読み込む．文字列は， ・n 文字目まで ・改行まで ・ファイルの終わりまで のいずれかの条件が満たされるまで読み込まれる．エラーまたはファイルの終わりならば NULL を返す	if (NULL == fgets (buff, 255, fp)) … fp で示すファイルから 1 行もしくは 255 文字を配列 buff に読み込む．関数の返値が NULL ならば，続く実行文を実行する
gets cf2	char *gets (b); char *b;	標準入力から 1 行(改行まで)を配列 b に読み込む	gets (buff); 標準入力から配列 buff に 1 行読み込む．buff の終わりには"0x00" が入る
fputs	int *fput (b, fp); char *b; FILE *fp	fp で示すファイルに配列 b の内容を書き出す	fputs (buff, fp); fp で示すファイルに配列 buff の内容を書き出す
puts	int puts (b); char *b;	標準出力に配列 b の内容を書き出す．このとき，ヌル文字(0x00)を改行(¥n)に置き換える	puts ("ABC"); 標準出力に"ABC" を書き出し，改行する

cf1:　BDS C の場合，「fgets (b, fp);」となり n バイトの指定はできない
cf2:　MSX-C，LSI C では，「gets (b, n);」となり読み込むバイト数 n を指定する

■フォーマット化入出力関数

関数名	書　式	機　能	使　用　例
printf	int printf (format [, list]); char *format;	format で指定する文字列を標準出力に出力する	printf ("%d", a); 変数 a の内容を 10 進数で表示する
fprintf	int fprintf (fp, format [, list]); FILE *fp; char *format;	format で指定する文字列を fp のファイルに書き出す	fprintf (fp, "%10s", a); 配列 a の文字列を 10 カラムにそろえて fp で示すファイルに書き出す
sprintf	sprintf (b, format [, list]); char *b; char *format;	配列 b に format で指定する文字列を書き出す	sprintf (buff, "%f", a); 変数 a を浮動小数点形式で配列 buff に書き出す
scanf	int scanf (format [, list]); char *format;	標準入力から format で指定する形式で変数 list に読み込む	scanf ("%s", a); 標準入力から文字列として配列 a に読み込む
fscanf	int fscanf (fp, format [, list]); FILE *fp; char *format;	fp のファイルから format で指定する形式で変数 list に読み込む	fscanf (fp, "%10d %10s", a, b); 変数 a, b に fp で示すファイルから，10 カラムにフォーマットした 10 進数と文字列を読み込む
sscanf	int sscanf (b, format [, list]); char *b; char *format;	配列 b から format で指定する形式で変数 list に読み込む	sscanf (buff, "%f", a); 変数 a に配列 buff から浮動小数点形式で数値を読み込む

注意：1．[　]は，省略可能
　　　2．format と list で指定する文字列については，「printf 関数の使い方」(210ページ)を参照のこと

■その他の関数

関数名	書　式	機　能	使　用　例
exit cf1	exit (s); int s;	プログラムを終了する．引数 s は通常， 正常終了…0 エラー　…0 以外 を指定する	exit (0); プログラムを終了し，オペレーティング・システムに戻る
atoi	int atoi (s); char *s;	文字列 s を整数に変換する	gets (string);　i = atoi (string); 標準入力から取り込んだ文字列を整数に変換して変数 i に代入する

cf1:　引数 s の扱いは，C 言語により異なるのでマニュアルを参照のこと

注意：C 言語によっては，「stdio. h」以外のヘッダファイルを取り込む必要がある．使用にあたっては，各マニュアルを参照のこと

練習問題の答え

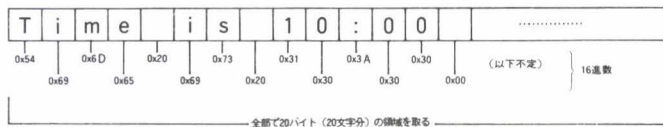
■ 第2章 ■

1.

```
 1: main() ………MSX-C, LSI Cは，1行目の前に「#include  <stdio.h>」を追加
 2:      {
 3:      int a;
 4:
 5:      a = 16161; ………10進数で書いた場合
 6:
 7:      printf("¥nNumber is %x in hexadecimal.", a);
 8:      printf("¥nNumber is %d in decimal.", a);
 9:      printf("¥nNumber is %o in octal.", a);
10:      }
```

2.

```
printf("*** This line has %d bugs. ***", num);
                                      ↑      ↑
                            「"」が抜けている  最後に「;」がない
```

3.

4. これはC言語によって解釈が違います．多くのC言語ではprintfの「" "」のなかの「/＊　＊/」はコメントとして扱われないようです．

■ 第3章 ■

1.

```
                   :
 4:
 5:      int i, a;
 6:      for(i = 0 ; i < 20 ; i++)
 7:           {
 8:           switch(a)
 9:                {
```

```
10:              case    1: printf("*** Devil Appeared !¥n");
11:                         break;
12:              case    2: printf("*** There are many angels !¥n");
13:                         break;
14:              case    4: printf("*** The Dragon's sun is dead.¥n");
15:                         break;
16:              default:   printf("*** You are dead Good-bye !¥n¥n");
17:              }
18:         }    :
19:              :
```

2.

```
 1: static char number[21] = "12345678901234567890"; ………MSX-C, LSI Cは,1行目の前に「#include <stdio.h>」を追加
 2:                             :………BDS Cの場合は,「1: char *number;」に変更
 3: main()                                 5行目の後に「number="12345678901234567890";」を追加
 4:     {
 5:     int    i;
 6:
 7:     for(i = 0 ; i < 20 ; ++i)  ………for文を使う方が簡潔に書ける
 8:         {
 9:         printf("Number is %c¥n", number[i]);
10:         }
11:     }
```

第4章

1.

```
 1: main() ………MSX-C, LSI Cは,1行目の前に「#include <stdio.h>」を追加
 2:     {
 3:     int     a, *b;
 4:
 5:     a = 3250;
 6:
 7:     b = &a; ………変数bのポインタ(アドレス)をaと同じにする
 8:
 9:     printf(" a == b == %d¥n", *b);
10:     }

A>test↵
 a == b == 3250
```

2.

```
 1: main() ………MSX-C, LSI Cは,1行目の前に「#include <stdio.h>」を追加
 2:     {
 3:     unsigned int    address; ………RUN/C, BDS Cは「unsigned address;」に変更
 4:     char            *data;
 5:
 6:     for(address = 0 ; address <= 65535 ; ++address)
 7:         {
 8:         data = address;
 9:         printf("ADDRESS = %5x : DATA = %4x¥n", address, *data);
10:         }
11:     }
```

3. ◆inputd関数の仕様

引数	出力するコメントの文字配列のポインタ．型は「char [ ]」
返値	入力された値．型は「int」

◆ inputd 関数とその例題

```
 1: main()………MSX-C, LSI Cは,
 2:      {        「#include  <stdio.h>」を追加
 3:      printf("¥n Value = : %d¥n", inputd("Input value in decimal : "));
 4:      }
 5:       input関数
 6: int inputd(comment)
 7:      char    comment[];
 8:      {
 9:      int ans;
10:
11:      printf("%s", comment);
12:      scanf("%d", &ans);
13:      return(ans);
14:      }

A>test↵
Input value in decimal : 300 ↵

 Value = : 300
```

printf関数のなかで直接inputd関数を呼び出している

〈補足〉
関数の引数として，以下の2つは同じ扱いになる
①「" "」で囲まれた文字列
②配列のポインタ
①の場合は，コンパイラが自動的に変数を定義し，そのポインタを関数の引数として渡たす

第5章

1.

```
 1: #include     <stdio.h>
 2: main()
 3:      {
 4:      FILE    *fp0,*fp1;
 5:      char    buff0[30], buff1[30];
 6:
 7:      printf("¥nSOURCE-FILE : "); scanf("%s", buff0);
 8:      printf("¥nDEST. -FILE : "); scanf("%s", buff1);
 9:
10:      if(NULL == (fp0 = fopen(buff0, "r"))) mesg0();
11:      if(NULL == (fp1 = fopen(buff1, "w"))) mesg1();
12:
13:      while(!feof(fp0)) fputc(fgetc(fp0), fp1);
14:
15:      fclose(fp0); fclose(fp1);
16:      }
17:
18: mesg0(){ printf("Cannot open SOURCE-FILE¥n"); exit(1);}
19: mesg1(){ printf("Cannot open DEST. -FILE¥n"); exit(1);}
```

＊注 ここでは，やっていることを1行で書いています
こんな書き方もできるという見本です

関数mesg 0 を呼ぶ

関数mesg 1 を呼ぶ

BDS Cは，ファイルの扱いが異なるので注意

ここでは読み出しと書き込みを一文で書いてしまっている

feof関数のない処理系は，関数の返値で判断する

メッセージを表示してプログラムを終了させる関数

2.

```
 1: #include     <stdio.h>
 2: main()
 3:      {
 4:      FILE     *fp0,*fp1;
 5:      char     buff0[30], buff1[30];
 6:      char     c[5]; ………消去する文字を入れるバッファを大きめに確保している
 7:
 8:      printf("¥nSOURCE-FILE : "); scanf("%s", buff0);
 9:      printf("¥nDEST. -FILE : "); scanf("%s", buff1);
10:      if(NULL == (fp0 = fopen(buff0, "r"))) mesg0(); } BDS Cでは，ファイルの扱いが異なるので注意
11:      if(NULL == (fp1 = fopen(buff1, "w"))) mesg1(); }
12:      printf("¥ndelete character = "); scanf("%s", c);
13:
14:      while(1) ………無限ループを作っている
15:           {
16:           int     cx;          ……… fgetc関数のない処理系は，getc関数を使う
17:           if(EOF == (cx = fgetc(fp0))) break; ………文字を読み込み，ファイルの終わりかどうかを判断
18:           if(c[0] != cx) fputc(cx, fp1); ………fputc関数のない処理系は，putc関数を使う
19:           }            ………… 消去する文字でなければファイルへ書き出す
20:      fclose(fp0); fclose(fp1);
21:      }
22:
23: mesg0(){ printf("Cannot open SOURCE-FILE¥n"); exit(1);}
28: mesg1(){ printf("Cannot open DEST. -FILE¥n"); exit(1);}
```

■ 第6章 ■

1. 答えはとくに示しませんので各自やってみてください.

2.

```
 1: #include     <stdio.h>
 2: main()
 3:      {
 4:      FILE    *fp;
 5:      char     fname[64];
 6:      char     line[256];
 7:      int      i;
 8:
 9:      printf("¥nInput file-name : ");
10:      gets(fname); ………MSX-C, LSI Cでは「gets(fname, 64);」に変更
11:
12:      if(NULL == (fp = fopen(fname, "w"))) exit(1);
13:                  ………BDS Cではファイルの扱いが異なるので注意
14:      for(i = 1 ; ; i++) ………カウンタを1つずつ増加させるための無限ループ
15:           {
16:           printf("LINE %3d : ", i); } 行番号を表示し，文字列を取り込む
17:           fgets(line, 256, stdin);  }
18:  BDS Cでは指定できない………        ← 標準入力(キーボード)の構造体FILEへのポインタ
19:           if ('.' == line[0]) break;
20:                  ……… 最初の1文字が"."(ピリオド)ならば入力終了
21:           fputs(line, fp);
22:           }
23:      fclose(fp);
24:      }
```

索引

記号

ア

カ

サ

タ

ナ

【参考文献】

●「C言語入門」ハンコック，クリーガー共著　アスキー出版局監訳　アスキー出版局
●「プログラミング言語C」カーニハン，リッチー共著　石田晴久訳　共立出版
●「C言語プログラミング入門 UNIX版」西田，五月女共著　啓学出版
●「UNIX System Vユーザ・リファレンス・マニュアル」日本ユニソフト㈱訳　共立出版
●「Microsoft Cユーザーズガイド，ランゲージリファレンス」マイクロソフト㈱
●「Microsoft C ランタイム ライブラリ リファレンス」マイクロソフト㈱

【筆者紹介】

三田 典玄（みた のりひろ）

1957年生まれ.

オーディオ関連会社,コンピュータ・エンジニアを経て,1986年2月 Coredump Co.,Ltd.を設立.

現在,その専務として各所を飛び回り多忙な日々を送っている.

また,UNIXマシンを個人で所有し,パソコン・ネットワーク(Personal Unix Net)を開設中.会員も150人を越え好評を得ている.

入門C言語
アスキー・ラーニングシステム①入門コース

1986年4月25日　初版発行
1988年12月21日　第1版第14刷発行
定価1,500円

著　者　三田典玄（みたのりひろ）
発行者　塚本慶一郎
発行所　株式会社アスキー
　〒107 東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111(大代表)
情報TEL (03)498-0299(ダイヤルイン)
出版営業部TEL (03)486-1977(ダイヤルイン)

編集担当　佐藤英一
表紙担当　郷　啓子
CTS　福田工芸株式会社
印刷　東京書籍印刷株式会社

ISBN4-87148-195-6 C3055 ¥1500E